FlexPref: A Framework for Extensible Preference Evaluation in Database Systems

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

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA
{[1]justin, [2]mokbel,[3]khalefa}@cs.umn.edu

***Abstract*— Personalized database systems give users answers tailored to their personal preferences. While numerous preference evaluation methods for databases have been proposed (e.g., skyline, top-k, k-dominance, k-frequency), the implementation of these methods at the *core* of a database system is a double-edged sword. Core implementation provides efficient query processing for arbitrary database queries, however this approach is not practical as *each* existing (and future) preference method requires a custom query processor implementation. To solve this problem, this paper introduces FlexPref, a framework for extensible preference evaluation in database systems. FlexPref, implemented in the query processor, aims to support a wide-array of preference evaluation methods in a single extensible code base. Integration with FlexPref is simple, involving the registration of only *three* functions that capture the essence of the preference method. Once integrated, the preference method "lives" at the core of the database, enabling the efficient execution of preference queries involving common database operations. To demonstrate the extensibility of FlexPref, we provide case studies showing the implementation of three database operations (single table access, join, and sorted list access) and five state-of-the-art preference evaluation methods (top-k, skyline, k-dominance, top-k dominance, and k-frequency). We also experimentally study the strengths and weaknesses of an implementation of FlexPef in PostgreSQL over a range of single-table and multi-table preference queries.**

I. Introduction

Embedding preferences in or on-top of databases has helped realize non-trivial applications, ranging from multi-criteria decision-making tools to personalized databases [1]. Preference queries give users interesting answers by evaluating their personal wishes according to a certain preference method. In the literature, there exist a large number of preference evaluation methods, including top-k [2], skylines [3], hybrid multi-object methods [4], k-dominance [5], k-frequency [6], ranked skylines [7], k-representative dominance [8], distance-based dominance [9], ϵ-skylines [10], and top-k dominance [11]. In general, the point of proposing new preference methods is to challenge the notion of "best" answers. Since the concept of "best" is subjective, there is theoretically no limit to the number of new preference methods that can be proposed. Given the large number of preference methods already proposed, and likely to be created in the future, a fundamental issue behind *each* method is how it can handle arbitrary queries in a database management system (DBMS) that may contain selection, aggregation, and/or join operations.

This work is supported in part by the National Science Foundation under Grants IIS0811998, IIS0811935, CNS0708604, and by a Microsoft Research Gift

The most common approach for preference evaluation in database systems is the *on-top* approach where the preference method is realized as either a stand-alone program or as a database user-defined function. This approach treats the DBMS as a "black box", where the preference evaluation method is completely decoupled from the database, and hence not concerned with internal database operations (e.g., joins) necessary to retrieve the data (e.g., see [4], [5], [6], [7], [8], [9], [10], [11]). The main advantage of this approach is its simplicity as it only requires the implementation of the preference evaluation method in a separate code base outside the core database engine. However, the efficiency of this approach is limited as it cannot interact with database internal operations, and hence cannot optimize the database query processor to be advantageous to the preference evaluation method [12], [13]. Furthermore, preference evaluation methods may be created assuming that data exists in a specific format (e.g., non-standard index), unaware of how data is physically stored or retrieved from the database.

A much more efficient approach for preference evaluation in database systems is the *built-in* approach that tightly couples preference evaluation with the query processor by creating optimized, customized, database operations (e.g., selection, aggregation, and join) for *each* preference method. The efficiency of this approach over the on-top approach is obvious from the extensive work of injecting ranking and top-k queries inside the database engine for selection queries [14], [2], join queries [15], and sorted list access [16], [17]. However, it is not practical to develop a database system that handles *each* existing and future preference method. Given the amount of effort needed to inject *top-k* operations in database systems [18], it would be hard to replicate this effort for each preference method. From a practical standpoint, supporting each distinct preference method in this manner is impossible for actual database systems, commercial or otherwise.

In this paper, we present FlexPref; an *extensible* framework for preference evaluation in database systems. FlexPref represents a centrist approach to preference implementation that combines the simplicity of the *on-top* approach with the efficiency of the *built-in* approach. The simplicity of FlexPref comes from the fact that integrating a new preference method involves the registration of only three functions that capture the essence of the preference method. The efficiency of FlexPref comes from the fact that once a preference method is integrated with the system, it "lives" at the core of the database,

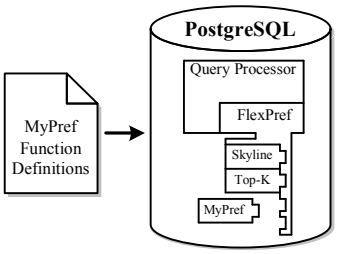


Fig. 1. Flexpref Architecture

coupled with the database engine and its query processor, enabling the efficient execution of preference queries involving common database operations.

As depicted in Figure 1, FlexPref is implemented inside the PostgreSQL [19] query processor, and is extensible to arbitrary preference methods. Injecting a preference method into the database requires the implementation of only three functions (outside PostgreSQL) that are then registered with FlexPref. These functions are designed to: (a) specify rules for when a tuple is "preferred" and (b) define rules for how items are added to a current set of preferred objects. Requiring the implementation of only three functions, preference evaluation methods are easier to implement in FlexPref compared to the *built-in* approach. In fact, FlexPref requires orders of magnitude less code. For example, implementing a simple single table skyline evaluation algorithm from scratch in PostgreSQL takes an order of 2,000 lines of code, while with FlexPref embedded in PostgreSQL, skyline implementation is on the order of 300 lines of code.

FlexPref results in efficient realization of preference methods inside the database engine, similar to that of the *built-in* approach. The main idea is that we tightly couple FlexPref with database operations in a customized manner. For example, we design single table access, join, and sorted list access operations that are customized to FlexPref. Then, any preference method registered with FlexPref is seamlessly integrated with the FlexPref framework, that is in turn coupled with the query processor. As depicted in Figure 1, it is important to note that *only* FlexPref touches the query processor while each new preference method is "plugged into" the framework. FlexPref raises two fundamental questions regarding the efficient execution of *arbitrary* preference queries: (1) *Is FlexPref more efficient than the on-top approach?* The answer is yes; coupling database operators with general preference criteria implies that a query processor can be optimized to perform early pruning by disregarding data that has no chance of being in a preferred answer set. Such an optimization is not possible with the *on-top* approach. (2) *Is FlexPref more efficient than the built-in approach?*. The answer, invariably, is no. Implementing specialized database operations for a specific preference method (e.g., top-k join) will always be more efficient than the generalized extensible case of FlexPref. However, it is impractical to have specialized implementations for *each* preference method. We equate this argument to previous research comparing generalized indexes (e.g., Gist [20]) to that of specialized indexes (e.g., B-tree [21], R-Tree [22]).

We demonstrate the functionality of FlexPref through three database operations, namely, *single table access*, *joins*, and *sorted list access*, that are designed to handle any arbitrary preference method integrated in FlexPref. We also provide case studies for integrating five non-trivial, state-of-the-art preference methods within FlexPref, namely, *Skyline* [3], *Top-k* [2], *Top-k dominating* [11], *K-dominance* [5], and *K-frequency* [6]. FlexPref has the potential to provide further functionality beyond the operations discussed in this paper, as it lays the groundwork for further non-trivial, extensible support for preference evaluation in databases, such as query optimization techniques, uncertain data processing, and indexing. The idea is that any new functionality will need to be realized *only once* for the FlexPref framework, instead of re-implementing it for *each* preference method. We experimentally evaluate the strengths and weaknesses of FlexPref, implemented in PostgreSQL, using our three main operations and five case studies, compared to *on-top* and *built-in* approaches.

The rest of this paper is organized as follows. Section II covers related work. Section III describes the usage of FlexPref. The FlexPref generic functions are described in Section IV. Section V covers preference evaluation in FlexPref through three main database operations. Five case studies of FlexPref are discussed in Section VI. Experimental evaluation of FlexPref is given in Section 7 while Section 8 concludes this paper.

II. Related Work

Preference methods. Many methods have been proposed for evaluating user preferences over relational data. The two methods receiving the most attention are skyline [3], [23], [24] and top-k [25], [2], [15], [26]. Other methods have been proposed that evaluate preference queries in a manner different to skyline and top-k, aiming to enhance the *quality* of the answer. Examples of these methods include, but are not limited to, hybrid multi-objective methods [4], k-dominance [5], k-frequency [6], ranked skylines [7], k-representative dominance [8], distance-based dominance [9], ϵ-skylines [10], and top-k dominance [11]. In this paper, we do not propose a new preference method. Rather, the goal of FlexPref is to provide a *single* generalized, extensible preference evaluation framework that allows the integration of *any* of these preference methods *inside* a database query processor.

Preference in databases. Much work has gone into embedding the notion of preference in database systems from both the *modeling* and *implementation* aspects. The *modeling* aspect is concerned more with the theoretical foundation of preference expressions over relational data [27], [28], [29], [30], [1], [31]. In some cases, the model provides rules that define how the model translates into traditional SQL queries. For example, query personalization [1], [32], [33] models preferences using a relational graph, where preferred attributes and relations are given a degree of interest score. Using this graph, SQL queries are injected with the top-k preferences derived from the graph. Meanwhile, PreferenceSQL [30] provides new SQL constructs for expressing preference by defining rules for combining preferences in a *cascading* or *pareto-accumulation* manner. Later work defined rules for

translating PreferenceSQL into traditional SQL queries [34]. Other work has explored modeling *contextual* preferences, where the objective is to evaluate preferences that change based on a user's situation [35], [36], [37].

In terms of preference method *implementation*, many proposed algorithms are not designed to integrate with ad-hoc relational queries involving joins, aggregation, etc [4], [5], [6], [7], [8], [9], [10], [11]. They are implemented "outside-the-box": completely outside the DBMS or as user-defined functions that sit *on-top* of a query plan. The closest work to ours investigates integrating preference evaluation algorithms within a database query processor. To this extent, there has been work integrating top-k preferences with selection [3], [2] and join queries [15], and skylines for join queries [15]. Conversely, we do not study *custom* implementations. FlexPref aims to support *any* preference method inside the database engine in a general, extensible manner. The FlexPref framework is completely novel in this regard.

III. Using FlexPref

In this section, we show how to: (a) register a new preference method in FlexPref, and (b) how to query a database system, e.g., PostgreSQL, that is equipped with FlexPref.

A. Adding A Preference Method to FlexPref

Adding a preference evaluation method to FlexPref requires the implementation of *three* functions outside the database engine. The details of these functions are covered in Section IV. Once implemented, the preference method is registered using a `DefinePreference` command, formally:

```
DefinePreference [Name] WITH [File]
```

The *name* argument is the name of the preference method, while the *file* argument specifies the file containing the function definitions. `DefinePreference` compiles the preference code into our framework. This process is depicted in Figure 1 for a preference method "MyPref".

B. Querying FlexPref

Once a preference method is registered with FlexPref, it can be used in database queries immediately. FlexPref requires the extension of the SQL syntax in order to select the appropriate preference methods and specify their objectives. In this section, we will first describe the general skeleton of SQL queries in FlexPref, and then describe the specific arguments for our five case studies of preference methods.

1) Query Skeleton: FlexPref adds a `Preferring` and `Using` clause to conventional SQL in order to issue preference queries. A typical query in FlexPref is:

```
Select [Select Clause]
From [Tables]
Where [Where Clause]
Preferring [Preference Attributes]
Using [method] With [Parameter]
Objectives [Objective]
```

Here, the method (with objectives) specified in the `Using` clause is responsible for selecting the preference evaluation method to be applied over the attributes given in the `Preferring` clause.

2) Five Case Studies: Using the query skeleton of FlexPref, we now give use case examples for five state-of-the-art preference methods, namely, *skyline* [3], *top-k* [2], *top-k dominating* [11], *k-dominance* [5], and *k-frequency* [6]. These preference methods are used throughout the rest of this paper to demonstrate the functionality of FlexPref.

Case Study I: 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. An example query using the skyline method is:

```
Select * From Restaurant R Preferring
  R.price d1 AND R.dist d2 AND R.rating d3
Using Skyline
With Objectives MIN d1, MIN d2, MAX d3;
```

This query will evaluate the skyline of restaurant data, where the preference objectives require minimizing both price and distance attributes, while maximizing rating.

Case Study II: Top-k dominating. The *top-k dominating* method ranks each object Q based on how many other objects it dominates, and returns the k objects with the highest score. Given the same preference attributes as the previous query, the `Using` clause for top-k dominating is:

```
Using Top-K-Domination With K=2
  Objectives MIN d1, MIN d2, MAX d3;
```

Here, the `Using` clause specifies that: (1) K=2 answers are required and (2) Preference is based on minimizing both price and distance attributes, while maximizing rating.

Case Study III: K-Dominance. The *k-dominance* method *redefines* the traditional skyline dominance definition to consider only k dimensional subspaces, where k is less than or equal to the total number of preference attributes. The `Using` clause for k-dominance is:

```
Using K-Dominance With K=2
  Objectives MIN d1, MIN d2, MAX d3;
```

For this case, the minimize/maximize objectives are similar to that of top-k domination and skylines. However, K specifies the *number of dimensions* used to check for dominance between objects. This is in contrast to usual definition of k that specifies the number of desired answers.

Case Study IV: K-Frequency. The *k-frequency* method ranks objects based on their dominance count in all possible dimensional subspaces, and returns the k objects with the minimal scores. The `Using` clause for k-dominance is:

```
Using K-Frequency With K=2
  Objectives MIN d1, MIN d2, MAX d3;
```

The objectives are the same as that of the top-k domination. However, k-frequency evaluates these objectives in a different manner in order to retrieve the "best" objects.

Case Study V: Top-k The *top-k* method scores each object by combining the object's attributes using a monotonic ranking function (e.g., summation) that returns a single real value. The k objects with the best scores are considered preferred objects. The `using` clause for the top-k method is:

```
Using Top-K With K=2
  Objectives MIN F(d1,d2,d3);
```

In this clause, K=2 answers are required, while the objective is to minimize an object score using monotonic ranking function F combining preference attributes $d1$, $d2$, and $d3$.

IV. FLEXPREF GENERAL FUNCTIONS

This section provides the details of the *three* general functions necessary to implement a preference method in FlexPref. To register a certain preference method, e.g., a *skyline*, the user needs to implement these three functions and populate them in the core of FlexPref using the `DefinePreference` command described in Section III-A. These functions are robust, as they can be used by FlexPref for various efficient query processing techniques as will be described in Section V. We also give five case studies of how these functions can be realized for various preference methods in Section VI.

Before discussing the three functions, we describe two definitions that will be used by our functions and the query processing techniques in Section V

- **`#define DefaultScore`**: Each object in FlexPref is associated with a score that is internal to the underlying preference method. It is provided by FlexPref so that the preference method may track the "quality" of each tuple during execution. Defining a default score ensures that each object is assigned a value.
- **`#define IsTransitive`**: Indicates whether the method is transitive or not. That is, given objects a, b, and c, if a is qualitatively "better" than b, and b is "better" than c, then a is always "better" than c. Knowledge of transitivity leads to efficiency, as FlexPref can discard objects during query execution if transitivity holds.

Three general functions need to be implemented by each preference method to be registered with FlexPref.

- **`PairwiseCompare(Object P, Object Q)`**: Given two data objects P and Q, *update* the score of P and return 1 if Q can *never* be a preferred object, -1 if P can *never* be a preferred object, 0 otherwise.
- **`IsPreferredObject(Object P, PreferenceSet S)`**: Given a data object P and a set of preferred objects S, return *true* if P is a preferred object and can be added to S, *false* otherwise.
- **`AddPreferredToSet(Object P, PreferenceSet S)`**: Given a data object P and a preference set S, add P to S and remove or rearrange objects from S, if necessary.

These functions break down preference evaluation into a set of modular operations that need *not* be aware of query processor specifics. FlexPref abstracts preference evaluation into two main operations: (1) pairwise comparison of two objects (`PairwiseCompare`) and (2) comparison of an object with one ore more objects in the current preference set (`IsPreferredObject`). Though these two functions could be enough for preference registration, FlexPref also provides a third manipulation function, `AddPreferredToSet`, to allow maximum flexibility and efficiency for the implemented preference method. For example, each preference method may keep the set S sorted in a manner advantageous to the execution of `IsPreferredObject`. For preference methods that require k answers, `AddPreferredToSet` has the ability to add a new object while removing an old object to ensure that only k objects exist in S.

In terms of the scope, FlexPref is able to support a range of qualitative (i.e., skyline) and quantitative (e.g., top-k/ranking) preference methods, as will be discussed in Section VI. If any preference method can reliably define answers by comparing tuples pairwise and/or comparing a tuple to a current preferred set, then it is supported by FlexPref. In general, we envision any preference model/method whose semantics define imperative steps that determine whether a tuple is preferred will work in FlexPref.

V. PREFERENCE EVALUATION IN FLEXPREF

This section explores the details of preference evaluation in FlexPref that uses the three main functions, `PairwiseCompare`, `IsPreferredObject`, and `AddpreferredToSet`, described in Section IV. We will first present single table access, i.e., selection queries over single table, in FlexPref in Section V-A. Then, in Section V-B we discuss how FlexPref is optimized to process multi-table queries, i.e., join queries. Finally, we discuss a query case when the input is represented as a set of sorted lists (i.e., indexes) in Section V-C. Without loss of generality, the examples throughout the rest of this paper use numeric data. However, FlexPref is compatible with methods for preference evaluation over other data types (e.g., partially-ordered domains [38]).

A. Single-Table Access

Single table access selects a set of preferred objects from a single table. All the query examples given in Section III refer to a single table access where the objective is to retrieve the set of preferred restaurants, according to a certain preference criteria, where all the information is stored in a single table R. We propose a block-nested loop (BNL) algorithm to execute single-table preference evaluation. We assume data is *not* indexed, thus a BNL algorithm is necessary (index access is discussed in Section V-C). The main idea is to compare tuples pairwise while incrementally building a preferred answer set. During execution, a data object P may be found to be dominated (i.e., guaranteed *never* to be a preferred answer). If the underlying preference method is *transitive*, P is immediately discarded and not processed further, thus leading to more efficient execution.

Algorithm 1 outlines the main steps of single-table preference evaluation in FlexPref. Underlined functions and definitions refer to those functions and definitions that should be implemented separately for each preference method registered with FlexPref, as described in Section IV. While simple, this single execution framework is very powerful as it can accommodate myriad preference evaluation methods. As proof, Section VI will cover the implementation of five state-of-the-art preference methods in this framework with execution examples. It is important to note here that Algorithm 1 is generic in the sense that it executes without knowledge of the general preference function details.

Algorithm 1 Single Table Access in FlexPref

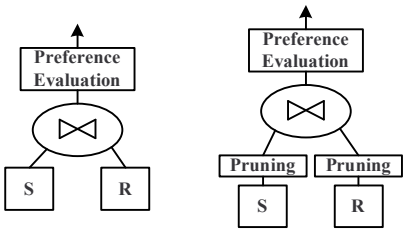
```
1:  Function SingleTableAccess(TableReference T)
2:  Preference Set S ← Φ
3:  for each Object P in T do
4:      P_score ← DefaultScore
5:      for each Object Q in T do
6:          cmp ← PairwiseCompare(P,Q)
7:          if cmp = 1 then
8:              if Q ∈ S then remove Q from S
9:              if IsTransitive then discard Q from T
10:         end if
11:         if cmp = -1 then
12:             if IsTransitive then discard P from T
13:             Read next object P (go to line 3)
14:         end if
15:     end for
16:     if IsPreferredObject(P,S) then AddPreferredToSet(P,S)
17: end for
18: return S
```

The input to Algorithm 1 is a reference to a single database table T while the output is the final set of preferred objects S. The algorithm begins by initializing the preference set S to empty. Next, we loop over table T in a block-nested fashion. Object P is read in the outer loop, where definition `DefaultScore` assigns its initial score, while object Q is read in the inner loop. Each pair of objects P and Q are compared pairwise using the generic function `PairwiseCompare`, where the score of object P is updated accordingly. If `PairwiseCompare` returns 1 (i.e., Q can never be a preferred object), and Q currently exists in the preferred set S, then Q is removed from S. Further, if the preference method is transitive, Q is discarded from table T mainly for efficiency sake. Due to transitivity, an object that is dominated by Q is also dominated by P, thus there is no need to track Q. In the case that the underlying preference method is not transitive, we must still consider Q as it may invalidate other objects with which it has not yet been compared. On the other hand, if `PairwiseCompare` returns -1, then P can never be a preferred object. If transitivity holds, P is discarded from table T and the next object in the outer loop is read immediately. The argument here is similar to the case of removing Q should the underlying preference function be transitive. Finally, If object P is not discarded in the inner-loop, we call `IsPreferredObject` to verify if P is part of the preference answer. As we will see in Section VI, this is usually a very simple function that performs an $O(1)$ check based on the properties of P and S without the need to iterate over S. If this function returns *true*, P is added to S. The algorithm concludes by returning S after the block-nested loop execution finishes.

Performance factors. The complexity of the single-table access function is mainly influenced by the preference method's transitive property. If the method exhibits transitivity, the complexity of Algorithm 1 is $O(n)$ in the best case and $O(n^2)$ in the worse case. If the method is *not* transitive, clearly Algorithm 1 is $O(n^2)$. Of course, complexity is also influenced by the efficiency of the generic function implementation(s).

B. Multi-Table Access

The join operation is one of the most common, and expensive, operations of a DBMS query. Thus, joins will likely be common in preference queries when run on a DBMS. For example, consider a preference query asking about restaurant attributes price, distance, and rating, where price and distance are stored in the same table while the rating information is stored in a separate table that may be maintained by a third-party. In this case a join is necessary to fulfill the preference query. We now discuss how FlexPref handles join queries in an efficient manner. We do *not* assume that input data is sorted or stored in a particular index, thus our method is applicable to any join method (e.g., hash join, index-nested loop) over arbitrary join predicates. For presentation purposes, we discuss the case of a single binary join. However, the concept can be extended to m-way joins or a tree of binary joins.



(a) Naive join (b) Join in FlexPref

Fig. 2. Join operator

Figure 2(a) depicts a naive join-then-evaluate strategy to execute join preference queries for two tables R and S. The idea is to perform a complete join over the two input tables followed by a preference evaluation over the join result. This approach is inefficient, as it does not attempt to optimize the underlying join operator. FlexPref improves upon this naive execution strategy by using the preference criteria functions to *prune* tuples from the join input that are guaranteed not to be in the final answer. Figure 2(b) gives the FlexPref strategy for handling join queries where pruning is performed at all join inputs, then, a final preference evaluation is performed after joining the non-pruned tuples from each table. Pruning enhances join performance for two main reasons: (a) the amount of data to be joined from input tables is greatly reduced due to pruning the input data, and (b) the amount of data processed by the final preference evaluation after the join is reduced based on the multiplying factor of the join.

Algorithm 2 outlines the main steps for join operations in FlexPref where the input is two tables, R and S, to be joined while the output is the set of preferred objects. First, R and S are pruned by applying the single table access algorithm (Algorithm 1) to each join-key group in both tables. For example, consider the tables R and S in Figure 3(b). Assuming ID as a join key, table S contains four groups a, b, c, and d that contains three, three, one, and one tuple(s), respectively. Also, table R trivially contains four single-tuple groups. In this case, single table access would be performed *locally* over each group in S only, as R's groups contain only a single tuple. By doing so, and according to the underlying preference method, several tuples from each group in S could be pruned, and thus, do not need to be joined with tuples in R. This main idea is that we guarantee that these tuples cannot be preferred objects.

Pruning in Algorithm 2 works for the following reason. For

Algorithm 2 Multi-Table Access in FlexPref

```
1: Function MultiTableAccess(Table R, Table S)
2: R_pruned ← Prune R: apply function SingleTableAccess to each join-key group in R.
3: S_pruned ← Prune S: apply SingleTableAccess to each join-key group in S.
4: J ← Join over R_pruned and S_pruned using any join method.
5: return SingleTableAccess(J) /* Algorithm 1 */
```

each local join-key group, assume we have a set of preferred tuples P and non-preferred tuples N. We can say that tuples in P are qualitatively better than tuples in N in each join-key group. Given two tables R and S, the tuples in each join-key group of S will join with the *same* tuples in R. If the pruned tuples N in S are qualitatively worse than those in P, then tuples in N cannot become qualitatively better once joined with the same data as the tuples in P. Thus, the pruned tuples N can never be part of the preference query answer.

Once the pruning is done locally for each group in tables R and S, the rest of the entries in both tables, R_{pruned} and S_{pruned}, are joined together using any join method (Line 4 in Algorithm 2). Finally, FlexPref performs another single table access over the entire join result J. This is mainly because the non-pruned tuples form R and S have not yet been compared against each other. The result of this step is the final result of the preference query.

Performance factors. Performance of the multi-table preference join is affected by four main factors. (1) *Group By*, if objects are not already sorted by their join key, a group by operation is necessary. (2) The complexity of single-table algorithm as it operates on each group. Note that this operation can be done *in parallel*, as each join-key group is independent of one another. (3) The join algorithm. FlexPref can use any available join method, thus complexity is influenced by the most efficient join algorithm available. (4) The final single-table algorithm operating on the join result. In general, pruning before the join reduces the overall complexity of the join-then-evaluate operation. However, pruning is not *always* optimal. For instance pruning is not effective in a 1:1 join, as no tuple can be pruned. Section VII experimentally evaluates pruning effectiveness; we omit an in-depth analysis for space reasons.

C. Sorted List Access

Availability of sorted attribute lists, or ordered indexes (e.g., B+-tree), allow for efficient preference evaluation. The idea is that complete preference answer generation can be guaranteed after reading only a *portion* of the sorted data, thus reducing the I/O overhead compared to query processing over unsorted or non-indexed data. Figure 3(c) gives an example of this model where three-dimensional data is presented as three separate 2-ary lists where each list: (a) includes the tuple ID, and (b) is sorted based on its attribute. Note that sorted lists are also an abstraction of an ordered index, such as a B+-tree. Several techniques have been proposed in the literature to take advantages of the sorted lists in preference evaluation, e.g., *top-k* [15] and *skyline* [39]. As efficiency is one of the main goals of FlexPref, this section presents a generic

Algorithm 3 Sorted List Access in FlexPref

```
1: Function GeneralSortedAccess(Lists[n], M)
2: stop ← false; count ← 0
3: Partial Set P ← Θ
4: ∀_{i≤n} O[i]=Θ; F[i]=Θ
5: while stop = false do
6:     Read next tuple t from Lists[i] in round-robin order
7:     if first value read from List[i] then F[i]=t.val
8:     O[i]=t.val
9:     Update/Add tuples to P by combining t with existing tuples on t.id
10:    if count = M then
11:        stop ← StopSortedEval(P, O, F) ; count ← 0
12:    else
13:        increment count
14:    end if
15: end while
16: for each incomplete point q ∈ P do
17:    ∀_j s.t. j is an incomplete dimension of q, make random access to Lists[j] to complete q
18: end for
19: return SingleTableAccess(P) /* Algorithm 1 */
```

algorithm for preference evaluation dealing with sorted lists. Sorted list evaluation requires one extra function be registered with FlexPref.

The main idea behind sorted list access in FlexPref is as follows: (1) Tuples are read, one-by-one, from each list in a round-robin fashion. During this time, we incrementally create a list P of *partial* objects. This list stores the *id* of each tuple read so far, along with all values of the tuple that have been read. For example, consider reading one point from each table $D1$-$D3$ in Figure 3(c). In this case, P would store two objects: (a,5,_,3) and (b,_, 2,_). (2) Round-robin processing ends once a *stopping* condition is met. This condition is defined by an extensible function, provided by the preference function implementation. (3) After stopping, all partial tuples in P are "completed" by making a random access to each sorted list to fill in missing attributes. To complete an object (a,5,_,3), table $D2$ would be probed to form (a,5,3,3). (4) Finally, we perform a final preference evaluation over the list P.

To realize this idea, and to take advantage of sorted lists, FlexPref requires that each preference method defines the following function in addition to the three functions described in Section IV.

- **`StopSortedEval(Set P, Object O, Object F)`**: Given a set of partial objects P and two virtual objects O and F, return whether objects *currently* in P, once completed, are sufficient to perform preference evaluation.

The arguments O and F in `StopSortedEval`, store the last and first values read form each input list, respectively. For example, reading round-robin twice from each list $D1$ to $D3$ in Figure 3(c) will produce O=(7,3,3) and F=(5,2,3).

Algorithm 3 outlines the main steps of sorted list preference evaluation in FlexPref that takes as input a reference to n decomposed relations (*Lists*), sorted by attribute value (i.e., sorted lists). Each tuple in a list has two attributes, $t.id$ and $t.value$; we assume tuples are combined using $t.id$. The algorithm also takes as input an integer M, used for efficiency purposes to restrict the number of calls to function `StopSortedEval` to every M^{th} list access. Initialization sets

a boolean value *stop* to false, an integer *count* to zero, and partial set P along with virtual object O and F to *null* (Lines 2 to 4). Round-robin processing then starts, and continues until the boolean *stop* is set to true by `StopSortedEval`. A tuple t is read from the current round-robin input list i, and if it is the first tuple read from i, the i^{th} dimension of F is set to $t.value$. Meanwhile the i^{th} dimension of O is also set to $t.value$ (Lines 7 to 8). One or multiple tuples are then updated or added to P based on combining t with previously-read tuples based on $t.id$. If $count$ equals M, the boolean *stop* is set by calling extensible function `StopSortedEval` and $count$ is reset to 0. Otherwise, $count$ is incremented and round-robin processing continues (Lines 9 to 14). After round-robin processing, all objects in P are then "completed" by making a random access to the necessary lists(s) (Lines 16 to 18). Sorted access processing concludes by performing single-table preference evaluation over set P using the algorithm outlined in Algorithm 1 (Line 19).

The method used to build partial objects (Line 9 in Algorithm 3) can be implemented in many ways. For robustness, FlexPref builds partial objects by abstracting the operation as an m-way symmetric hash join join [40] between m decomposed (i.e., 2-ary) relations. The idea behind the symmetric hash join is to store a hash table for each input list i. When a tuple t is read from list i, it is hashed to table i using the value of its join attribute. Tuple t is then used to probe all other hash tables to produce partial (or full) objects.

Performance factors. Performance of sorted list access is influenced by two main factors. (1) The number of sorted accesses. In the worse case, a total of $m * n$ sorted reads are required (average case is $\frac{mn}{2}$), where m is the number of attributes and n the number of objects. (2) The efficiency of the generic `StopSortedEval` function implementation.

VI. Case Studies

In this section, we provide five case studies for injecting five state-of-the-art preference evaluation methods in FlexPref, namely, *skyline* [3], *top-k dominating* [11], *k-dominance* [5], *k-frequency* [6], and *top-k* [2]. We chose these five case studies carefully to cover a wide spectrum of preference methods. In particular, *skyline* represents transitive dominance-based preference methods, *k-dominance* represents non-transitive dominance-based preference methods, *top-k* represents ranking-based preference methods, *top-k dominating* represents preference methods that combine ranking-based and dominance-based preferences, and *k-frequency* represents methods that propose object rankings that do not require a specific function, but base their scoring on inherent properties of an object (e.g., attribute correlation and subspace search). Per our discussion in Section IV, FlexPref supports any preference method that can define answers by comparing tuples pairwise and/or comparing a tuple to a current preferred set.

For each preference method, we first describe its functionality. Then, we cover the implementation of the three general functions described in Section IV. The summary of all functions for the five case studies is given in Table I. Finally, we give illustrative examples, using Figure 3, for single table, multi-table, and sorted list access. Due to space, we provide and in-depth example for the skyline case study, and more summarized examples for subsequent case studies.

A. Case Study 1: Skylines

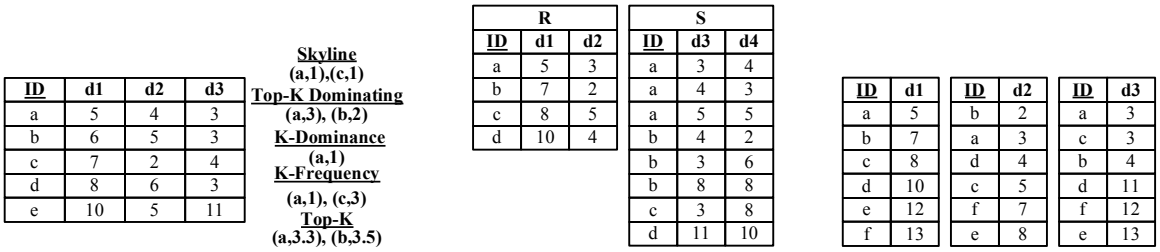
Given a dataset D, the objective of skyline preference evaluation [3] is to find the set of objects S that are *not dominated* by any other object in D. An object P is said to *dominate* an object Q if P is better than or equal to Q in all dimensions, and *strictly* better than Q in at least one dimension. For example, in Figure 3(a) object a dominates object e as it is better (i.e., less) in all the three dimensions.

1) General Function Implementation: A skyline implementation in FlexPref is given in the second column of Table I. Skyline evaluation does not rank objects. Thus, within our framework, the skyline score of an object P is binary and is set to one if P is not dominated, and zero otherwise. Initially, each object is assumed to be a skyline, thus, each object has a default score of one. Furthermore, skylines exhibit the transitive property, thus its transitive definition is set to *true*.

Function `PairwiseCompare` changes the score of P to zero *only* if it is dominated, and returns the appropriate value based on the dominance relation between P and Q, i.e., if P is dominated it cannot be a preferred object, and vice versa. Q's score is not updated in `PairwiseCompare` per the function definition given in Section IV. Function `IsPreferredObject` does not need the reference set S to determine if P is a preferred object. Instead, it only returns *true* if P_{score} is one, i.e., P was not dominated by any object. Function `AddPreferredToSet` simply appends P to the end of set S. The stopping condition for `StopSortedEval` can be based on previous research in distributed skyline query processing [39]. This condition is: *stop once there is a complete object Q in set P*. At this stopping point, the complete object Q is equal to, or dominates, the virtual object O. Furthermore, any new object added to P cannot be better than O, thus only objects currently in P are skyline candidates.

2) Preference Evaluation: **Single Table Execution.** Consider the three-dimensional data in Figure 3(a). First, object a is read, given a default score of 1, and compared pairwise with all other objects using `PairwiseCompare`. Function `PairwiseCompare` returns -1 when a is compared with object b, 0 when compared to c, and -1 when compared to d and e. Thus, b, c, and d are discarded from the data set. Since a is not dominated, function `PairwiseCompare` does *not* change a's score to 0. Thus, `IsPreferredObject` reports that a can be added to the preference answer. Object c is then read and also found to be a preferred answer (as it is not dominated by a, the only object left in the data set). After processing c, no objects are left in the data set and execution terminates. Objects a and c exist in the preference set, each with a score of one, as given in the skyline answer in Figure 3(a).

Multi-Table Execution. In Figure 3(b), pruning removes tuples (a,5,5) and (b,8,8) from table S prior to the join. These tuples are not skylines within their join-key groups.



ID	d1	d2	d3
a	5	4	3
b	6	5	3
c	7	2	4
d	8	6	3
e	10	5	11

Skyline
(a,1),(c,1)
Top-K Dominating
(a,3), (b,2)
K-Dominance
(a,1)
K-Frequency
(a,1), (c,3)
Top-K
(a,3.3), (b,3.5)

(a) Single-table example

R		
ID	d1	d2
a	5	3
b	7	2
c	8	5
d	10	4

S		
ID	d3	d4
a	3	4
a	4	3
a	5	5
b	4	2
b	3	6
b	8	8
c	3	8
d	11	10

(b) Arbitrary join data

ID	d1
a	5
b	7
c	8
d	10
e	12
f	13

ID	d2
b	2
a	3
d	4
c	5
f	7
e	8

ID	d3
a	3
c	3
b	4
d	11
f	12
e	13

(c) Sorted lists

Fig. 3. Case study example data

Furthermore, these tuples cannot possibly be skylines when joined with their corresponding tuples (a,5,5) and (b,7,2) in table R. For example, joined tuple (a,5,3,5,5) will at least be dominated by members of its same join group: both (a,5,5,3,4) and (a,5,5,4,3). Similarly, joined tuple (b,7,2,8,8) would be dominated by both (b,7,2,4,2) and (b,5,5,4,3).

Sorted Table Access. Round-robin processing can stop after five reads for the data in Figure 3(c). At this point, set P contains objects (a,5,3,3) and (b,7,2,_), while object O equals (7,3,3) and object F equals (5,2,3). Clearly, any new object added to P cannot be better than virtual object O due to sorted access, and any new object added to P will be dominated by the complete object a.

B. Case Study 2: Top-K Dominating

Given a data set D, the objective of top-k dominating preference evaluation [11] is to score each object P by its *dominance power*, i.e., the number of objects it dominates. Here, the dominance definition is the same as the skyline method. The preference answer contains the k objects with the highest score (i.e., the objects that dominate the most other objects). As an example, consider objects a and c in Figure 3(a). Object a has a score of three, as it dominates objects b, d, and e. Object c has a score of one as it only dominates e. Object preference is based solely on *dominance power*, thus non-skyline objects can be preference answers.

1) General Function Implementation: A top-k dominating implementation in FlexPref is given in the third column of Table I. Each object is given a default score of 0, and `IsTransitive` returns *true*. When function `PairwiseCompare` finds that P dominates an object Q, it increments P's score by one. An object can never be ruled out of the preference answer using pairwise comparison, since P's score must be calculated through comparison with all objects, thus `PairwiseCompare` always returns zero. Function `IsPreferredObject` returns true if P has a score superior to any of the current k objects in S, or if S contains less than k objects. Finally `AddPreferredToSet` adds P in sorted order in S, removing the old k^{th} object if applicable. A stopping condition for `StopSortedEval` can be: *stop once there are k complete objects in set P*. While it is possible that some *incomplete* objects in P will be superior to the complete k objects, this stopping condition at least ensures that the complete k objects dominate any objects not yet added to P. The unseen objects are equal-to or dominated by object O, which in turn is equal-to or dominated by each of the k complete objects.

2) Preference Evaluation: The top-k domination answer is given in Figure 3(a) assuming $k = 2$. In Figure 3(b), top-k domination pruning removes from S tuples (a,5,5) and (b,8,8) both with scores of zero. These pruned tuples are not in the top-2 in their *local* join-key groups. Meanwhile, sorted round-robin processing can stop after nine reads for the data in Figure 3(c). At this point, set P contains objects (a,5,3,3), (b,7,2,4), (c,8,_,3), and (d,_,4,_), while objects O equals (8,4,4) and object F equals (5,2,3).

C. Case Study 3: K-Dominance

Given a data set D and a value k, k-dominance preference evaluation [5] finds the set of objects S that are *not k-dominated* by any other object in D. k-dominant queries are similar in spirit to skyline queries, except for the relaxed notion of dominance: an n-dimensional object P is allowed to dominate another object Q on *any* $k \leq n$ dimensions. When $k = n$, a k-dominant query reverts to a skyline query. As an example, consider objects a and c in Figure 3(a). For $k = 2$, object a k-dominates object c since a is better in dimensions $d1$ and $d3$ (less is better). However, when $k = 3$ neither object dominates the other as in the case of skylines.

1) General Function Implementation: A k-dominance implementation in FlexPref is given in the fourth column of Table I. As k-dominance does not rank objects, each object can either have a score of one if it is not k-dominated, zero otherwise. Also, k-dominance is not transitive as *circular dominance* is possible: an object x can k-dominate an object y, y can k-dominate an object z, and z can k-dominate x. Thus, the `IsTransitive` property is set to *false*. The function `PairwiseCompare` changes the score of P to zero *only* if it is k-dominated by Q, and returns the appropriate value based on dominance relation between P and Q. `IsPreferredObject` does not need to reference set S to determine if P is a preferred object, and returns *true* if P's score is 1 (i.e., P is not k-dominated). Function `AddPreferredToSet` simply appends P to the end of set S. A stopping condition for `StopSortedEval` is: *stop once set P contains an object Q with at most $k-1$ incomplete dimensions, and Q k-dominates virtual object O, and O does not k-dominate Q (where the value ∞ is substituted for the incomplete dimensions of Q).* Having an object Q with at most $k-1$ incomplete dimensions ensures that it cannot be k-dominated on these incomplete dimensions by an object not yet added to P. Furthermore, since

TABLE I

REALIZING FIVE PREFERENCE METHODS IN FLEXPREF

	Skyline [3]	Top-k Dominating [11]	K-Dominance [5]	K-Frequency [6]	Top-K [3]
IsTransitive	Return *true*	Return *true*	Return *false*	Return *true*	Return *true*
DefaultScore	Return 1	Return 0	Return 1	Return 0	Return 0
PairwiseCompare	If P dominates Q, return 1. If Q dominates P, update P_{score} to 0 and return -1. Otherwise, return 0	If P dominates Q, increment P_{score} by 1. Return 0	If P k-dominates Q, return 1. If Q k-dominates P, set P_{score} to 0 and return -1. Otherwise, return 0.	Increment P_{score} based on the *distinct* sub-dimensions where Q dominates P. Return 0.	Return -1
IsPreferredObject	If object score P_{score} equals 1, return true. Otherwise return false.	If the cardinality of S is less than k or P_{score} is superior to the kth object's score in S, return *true*. Otherwise, return *false*.	If object score P_{score} equals 1, return true. Otherwise return false.	If the cardinality of S is less than k or P_{score} is superior to the kth object's score in S, return *true*. Otherwise, return *false*.	Assign a score to P_{score} using ranking function f. If the cardinality of S is less than k or P_{score} is superior to the kth object's score in S, return *true*. Otherwise, return *false*.
AddPreferredToSet	Add object P to the end of set S.	If S has a cardinality of k, remove the kth object from S. Add P to S in sorted order by P_{score}.	Add object P to the end of set S.	If S has a cardinality of k, remove the kth object from S. Add P to S in sorted order by P_{score}.	If S has a cardinality of k, remove the kth object from S. Add P to S in sorted order by P_{score}.
StopSortedEval	Stop once there is a complete object Q in set P	Stop once there are k complete objects in set P.	Stop once set P contains an object Q with at most $k-1$ incomplete dimensions, and Q k-dominates virtual object O, and O cannot k-dominate Q.	Stop once there is a complete object Q in set P.	Stop once there are k complete objects in P that have scores less than or equal-to a given threshold value T. T = MIN(f(O[1],F[2] $\cdots$,F[n]), f(F[1],O[2],$\cdots$,F[n]), f(F[1],F[2] $\cdots$,O[n])).

Q k-dominates virtual object O, but O does not k-dominate Q, then any object not yet added to P is guaranteed to be k-dominated by Q. Thus the k-dominant answer candidates must exist in P. We note that for multi-table execution, where tables R and S have contain d_R and d_S dimensions each, pruning computes the d_R and d_S-dominant answer, and then the k-dominant answer in the final (i.e., root) preference evaluation.

2) Preference Evaluation: The k-dominance answer is given in Figure 3(a) assuming $k = 2$. In Figure 3(b), pruning will remove from table S tuples (a,5,5) and (b,8,8), as they are k-dominated within their join-key groups. Similarly, round-robin processing can stop after five reads for the data in Figure 3(c), where set P contains objects (a,5,3,3) and (b,7,2,_) while object O equals (7,3,3) and object F equals (5,2,3).

D. Case Study 4: K-Frequency

Given a data set D, k-frequency preference evaluation [6] scores each object P by its *dominated subspaces*: the number of possible sub-dimensions in which P is dominated. The preference answer contains the k objects with the lowest score (i.e., the objects that are dominated in the least number of possible sub-dimensions). As an example, object a in Figure 3(a) has a score of one, since it can only be dominated in a single sub-dimension ($d2$ by object c). Meanwhile, object e has a score of 7, since it is dominated in all possible sub-dimensions by object a (i.e., $\{d1\}$, $\{d2\}$, $\{d3\}$, $\{d1, d2\}$, $\{d1, d3\}$, $\{d2, d3\}$, $\{d1, d2, d3\}$).

1) General Function Implementation: A k-frequency implementation in FlexPref is given in the fifth column of Table I. Each object is given a default score of zero, and `IsTransitive` returns *true*. Dominant sub-dimension counting must be performed carefully for k-frequency. For instance, in Figure 3(a) object c is dominated on overlapping dimensions by different objects. That is, c is dominated in sub-dimensions ($\{d1\}$, $\{d3\}$, $\{d1, d3\}$) by object a, and sub-dimension $\{d3\}$ by object d. Clearly, over-counting dominated sub-dimensions is an issue. Thus, function `PairwiseCompare` must have access to an extra data structure that stores the dominated sub-dimensions for each object P. Tracking these sub-dimensions ensures that an object is scored correctly, i.e., *distinct* sub-dimensions can be extracted and counted. So, Function `PairwiseCompare` updates P_{score} based on the *distinct* sub-dimensions where Q dominates P. An object can never be ruled out of the preference answer using pairwise comparison since P's score must be calculated through comparison with all objects, thus `PairwiseCompare` always returns zero. Function `IsPreferredObject` returns true if P has a score superior to any of the current k objects in S or if S contains less than k objects. `AddPreferredToSet` adds P in sorted order in S, possibly removing the old k^{th} object. `StopSortedEval` can use the same stopping condition as skylines. This condition guarantees that an interesting set of objects exists in P, as any object *not yet* added to P is guaranteed to be equal-to or dominated by the complete object Q. Thus, any object not in P is guaranteed to be dominated in all possible sub-dimensions, meaning that all unseen objects will have the same score. If there are not yet k objects in P when the stopping condition is met, then any arbitrary objects can be added to P as they have the same score.

2) Preference Evaluation: The k-frequency answer is given in Figure 3(a) assuming $k = 2$. In Figure 3(b), k-frequency pruning will remove from table S tuples (a,5,5) and (b,8,8), as they both have local scores of three, i.e., they are dominated in all possible subspaces. Round-robin processing can stop after five reads for the data in Figure 3(c). At this point, P contains objects (a,5,3,3) and (b,7,2,_), while object O equals (7,3,3) and object F equals (5,2,3).

E. Case Study 5: Top-K

Given a set of data D, top-k preference evaluation [2] scores each data object P using a monotonic ranking function f. The preference answer contains the k objects with the minimum

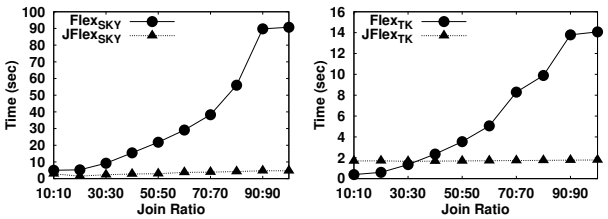
score. A monotone function f takes as input multiple attribute values of and object P and returns a single real number as its score. For example, for object a in Figure 3(a) and a monotone function f=($\frac{1}{10} * (d1 + d2) + \frac{4}{5} * d3$), a's score is 3.3.

1) General Function Implementation: A top-k implementation in FlexPref is given in the last column of Table I. Each object has a default score of zero, and `IsTransitive` returns *true*. Top-k does *not* rely on pairwise comparison since an object's score is determined using only its own attributes. Thus, `PairwiseCompare` return -1 by default; doing so will cause the inner loop to be broken in Algorithm 1, as this scan is not needed. `IsPreferredObject` computes object P's score using a monotonic ranking function f, and returns true if P has a score superior to any of the current k objects in S, or if S contains less than k objects. Function `AddPreferredToSet` adds P in sorted order in S, removing the old k^{th} object if applicable. A possible stopping condition for `StopSortedEval` is based on previous research that defines threshold score for efficient top-k joins over sorted lists [15]. Specifically, the condition is: *stop once there are k complete objects in P that have scores less than or equal-to a given threshold value T*. Threshold T is a lower-bound on the scores of any object not seen so far in set P, defined as MIN(f(O[1],F[2],$\cdots$,F[n]), f(F[1],O[2],$\cdots$,F[n]), f(F[1],F[2],$\cdots$,O[n])). That is, the minimum of the scores taken from combining the last value seen from each input with the first values read from every other input.

2) Preference Evaluation: The top-k answer is given in Figure 3(a) for $k = 2$, where we aim to minimize scores based on a ranking function that sums all attribute values. In Figure 3(b), pruning will removes from S tuples (a,5,5) and (b,8,8) with scores 10 and 16, respectively. These pruned tuples are not in the top-2 in their join-key groups. Round-robin processing can stop after 12 reads for the data in Figure 3(c). At this point, set P contains (a,5,3,3), (b,7,2,4), (c,8,5,3), (d,10,4,11), while O=(10,5,11), F=(5,2,3), and threshold T = 13.

VII. Experimental Evaluation

In this section, we experimentally evaluate the performance of FlexPref. Our experiments involve the following implementations: (1) The skyline, k-dominance, and top-k preference evaluation methods implemented in FlexPref according to the function definitions given in Section VI. These implementations are denoted Flex$_{SKY}$, Flex$_{KDOM}$, and Flex$_{TK}$, respectively. (2) The *custom* implementations of the skyline block nested loop operator (Cust$_{SKY}$) [3] and the two-scan K-dominance algorithm (Cust$_{KDOM}$) [5]. We also implemented the custom multi-relational skyline join operator (JCust$_{SKY}$) [41] in order to fairly evaluate our multi-relational preference execution framework. These custom implementations make for the fairest comparison against our framework as they do not assume sorted or indexed data. We note that for the case of top-k, no custom implementation exists that assumes completely unsorted/unranked input [18]. An exception to this claim is the case involving sorted list access, which we discuss in Section VII-B. We perform experiments for three query scenarios: (1) Multi-table access (Section VII-A), (2) Sorted list access (Section VII-B), and (3) Single table access (Section VII-C),



(a) Skyline (b) Top-K

Fig. 4. Multi-table FlexPref join

Our FlexPref framework, along with the custom skyline join algorithm [41], are implemented in the backend executor (query processor) of the PostgreSQL 8.3.5 open-source database [19]. All other algorithms are implemented as extensible user-defined functions in PostgreSQL. The user-defined function approach is the fairest implementation method for our counterparts, as it pushes the algorithms as close to the database as possible. The experiment machine is an Intel Core2 8400 at 3Ghz with 4GB of RAM running Ubuntu Linux 8.04. We use the generator specified in [3] to generate synthetic data sets for all experiments. Unless otherwise mentioned, the data contain six attributes, where the attribute values are generated independent of one another. We experiment with data set sizes ranging from 10K to 3M tuples. The value of k for the k-dominance preference is set at 4. For the top-k method, the default number of answers (k) is set to 20. Our performance metric is the *elapsed time* reported by the PostgreSQL EXPLAIN ANALYZE command.

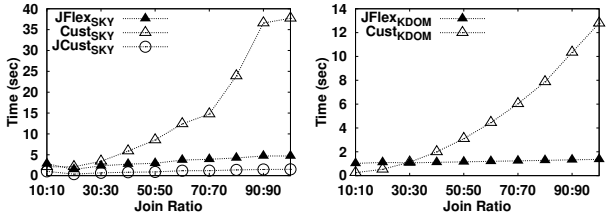
A. Multi-Table Access

In these experiments, we study the impact of the FlexPref multi-table preference evaluation framework that prunes join input tuples, and then compare the FlexPref implementations to Cust$_{SKY}$ and Cust$_{KDOM}$, as well as the custom skyline join algorithm JCust$_{SKY}$. The general SQL signature of the experiment query is:

```
Select * From T1, T2 WHERE T1.id=T2.id
Preferring T1.d1 AND T1.d2 AND T1.d3
  AND T2.d1 AND T2.d2 AND T2.d3
```

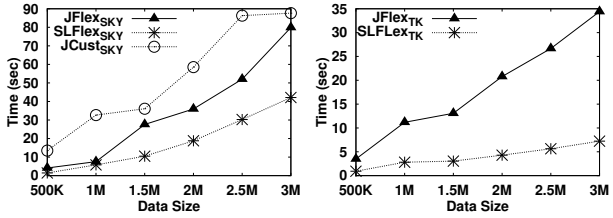
We omit the `using` clause as multiple preference methods are tested. The join is an m:m binary join where tables T1 and T2 contain three-attribute tuples, plus an id, while preference is evaluated over all six attributes. Each table contains 1K unique ids, with an equal number of tuples assigned to each join-key group. We increase the size of each table from 10K to 100K that increases the join ratio from 10:10 to 100:100, as well as the join result cardinality.

1) Effect of Pruning: This experiment studies the effect of pruning join inputs in FlexPref's multi-table execution. We study the skyline, and top-k implementations in FlexPref using the naive join approach (abbr. Flex$_{SKY}$, and Flex$_{TK}$) against the optimized pruned approach (abbr. JFlex$_{SKY}$ and



(a) Skyline (b) K-Dominance

Fig. 5. Multi-table: FlexPref vs. Specialized



(a) Skyline (b) Top-K

Fig. 6. Sorted list access

JFlex$_{TK}$). For space purposes, we do not discuss the k-dominance implementation, however, it exhibits similar behavior to the skyline case. Figure 4 gives the runtimes for the skyline (Figure 4(a)), and top-k (Figure 4(b)) methods. Clearly, pruning is beneficial to the FlexPref framework, keeping preference evaluation scalable for multi-table queries. For the skyline method, tuples are pruned throughout the progression of join ratios, reducing the workload of the join and final post-join preference evaluation. For the case of top-k (with default $k = 20$), pruning takes effect after the 20:20 ratio. For smaller ratios, no join input tuples can be pruned as join-key groups contain less than 20 tuples, thus pruning causes an overhead for these cases.

2) Comparison With Custom Algorithms: Given that pruning in FlexPref is beneficial to multi-table preference queries, we now compare the optimized skyline and k-dominance FlexPref implementations, JFlex$_{SKY}$ and JFlex$_{KDOM}$, against Cust$_{SKY}$ and Cust$_{KDOM}$ that must perform preference evaluation *after* the join (i.e., *on-top* of the query plan). We also compare FlexPref against the specialized skyline join operator [41], JCust$_{SKY}$). Figures 5(a) and 5(b) give the runtimes for skyline and k-dominance methods, respectively. These results clearly highlight the advantages of FlexPref. The optimized FlexPref implementations exhibit scalable behavior as the join ratio (and data size) increases. FlexPref is superior to the Cust$_{SKY}$ and Cust$_{KDOM}$ methods that represent an *on-top* approach for the multi-table case. Both Cust$_{SKY}$ and Cust$_{KDOM}$ cannot reduce the input to the join, thus must process the complete join result. Interestingly, JFlex$_{SKY}$ exhibits comparable performance to the *custom* skyline join JCust$_{SKY}$. These results are promising, and show that (1) FlexPref is clearly advantages for arbitrary DBMS queries compared to an *outside* (or *on-top*) and (2) competitive with *specialized* approaches for more sophisticated queries.

We do not compare our JFlex$_{KDOM}$ method against a custom k-dominance join algorithm, as none exist. The only possible implementation for k-dominance in the case of arbitrary multi-relational queries is to perform evaluation *on-top* of the query plan. This fact highlights the strength of FlexPref. Once registered with FlexPref, *any* preference method gains the advantages of being coupled with non-trivial database operations. This experiment highlights the efficiency gains by taking the general extensible approach of FlexPref.

B. Sorted List Access

This experiment studies the efficiency of the general sorted access preference evaluation algorithm outlined in Section V-C for the skyline and top-k methods. For space purposes, we do not plot the k-dominance experiment, however, it exhibits similar behavior to the skyline case. The general SQL signature of the experiment query is:

```
Select * From T1,...,T6
Where T1.id=T2.id=T3.id=T4.id=T5.id=T6.id
Preferring T1.d AND ... AND T6.d
```

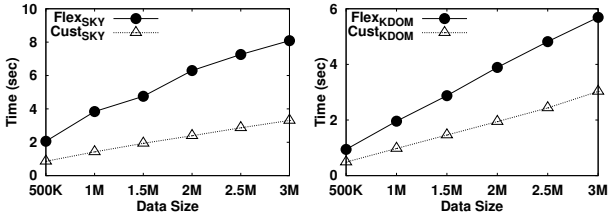
We again omit the `using` clause as multiple preference methods are tested. The join is 1:1 that combines six 2-ary tables T1-T6, each with a primary key id and attribute d; all tables are sorted on d. We compare the FlexPref optimized join implementation (JFlex$_{SKY}$ and JFlex$_{TK}$) to the FlexPref sorted list implementation for the skyline and top-k methods (SLFlex$_{SKY}$ and SLFlex$_{TK}$). For the skyline case, we also compare with JCust$_{SKY}$. We do not implement a custom join algorithm for top-k, as the FlexPref top-k sorted list impelementation actually reduces to an m-way version of the custom join specified in [15]. Figure 6 gives the runtimes for both skyline and top-k as the table sizes increase from 500K to 3M tuples. The results confirm the efficiency of the general sorted list access framework of FlexPref. As input size increases, the sorted list method makes use of the stopping condition in order to end I/O earlier during processing. Of course, The FlexPref join framework must read every input tuple in order to perform the full join. Interestingly, the custom skyline join JCust$_{SKY}$ shows poorer performance than both FlexPref implementations. This poor performance is due to JCust$_{SKY}$ needing to materialize *every* intermediate join result in the query tree in order to find a *global* skyline for the input to the subsequent join.

C. Single-Table Access

In this experiment, we study the performance of the skyline and k-dominance preference implementations for a single table access query. The general SQL signature is:

```
Select * From T
Preferring T.d1 AND ... AND T.d6
```

Figures 7(a) and 7(b) give the runtimes for the skyline and k-dominance methods, respectively, as the table cardinality is increased from 500K to 3M tuples. Both the FlexPref skyline and k-dominance implementations (Flex$_{SKY}$ and Flex$_{KDOM}$) show inferior performance to their counterpart custom implementations (Cust$_{SKY}$ and Cust$_{KDOM}$). Implemented as user-



(a) Skyline (b) K-Dominance

Fig. 7. Single-table preference evaluation

defined functions, both Cust_{SKY} and Cust_{KDOM} resemble a *specialized* approach for this experiment as they are designed to read data from a single, unsorted table. It is *not* the objective of FlexPref to win over these very *specialized* implementations for this case of single table access. The power of FlexPref appears in: (a) its support for optimizing more sophisticated queries, as we studied in previous experiments, where any preference method in FlexPref is coupled with non-trivial database operations, and (b) its practical approach to implementing a wide array of preference evaluation methods, which would require a great amount of effort without FlexPref. Having said so, Flex_{SKY} and Flex_{KDOM} display linear behavior similar to Cust_{SKY} and Cust_{KDOM}, as the FlexPref single-table access algorithm cuts its inner loop immediately when an outer object is found to be dominated, thus staying competitive with the customized algorithms [3], [5].

VIII. Conclusion

This paper presented FlexPref, a general framework for extensible preference evaluation. FlexPref is implemented in the query processor of a database, and supports a multitude of preference evaluation methods. Implementing a new preference method requires the registration of only *three* functions that capture its essence. Once integrated, the preference method "lives" at the core of the database, enabling the efficient execution of preference queries involving common database operations. We provided the details of how FlexPref is integrated into three database operations: single-table access, multi-table access involving arbitrary join patters, and sorted list access. We detailed the implementation of *five* non-trivial, state-of-the-art preference methods inside FlexPref. We also provided experimental evidence that verified the ability of FlexPref to provide efficient query support for arbitrary DBMS queries. FlexPref lays the groundwork for further general functionality support for preference evaluation in databases, including, but not limited to: uncertainty handling, indexing, and integration with aggregate operators.

References

[1] G. Koutrika and Y. Ioannidis, "Personalization of Queries in Database Systems," in *ICDE*, 2004.
[2] S. Chaudhuri and L. Gravano, "Evaluating Top-K Selection Queries," in *VLDB*, 1999.
[3] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," in *ICDE*, 2001.
[4] W.-T. Balke and U. Güntzer, "Multi-objective Query Processing for Database Systems," in *VLDB*, 2004.
[5] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang, "Finding k-Dominant Skylines in High Dimensional Space," in *SIGMOD*, 2006.
[6] ——, "On High Dimensional Skylines," in *EDBT*, 2006.
[7] J. Lee, G. won You, and S. won Hwang, "Personlized Top-K Skyline Queries in High-Dimensional Space," *Information Systems*, vol. 34, no. 1, pp. 45–61, 2009.
[8] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang, "Selecting Stars: The k Most Representative Skyline Operator," in *ICDE*, 2007.
[9] Y. Tao, L. Ding, X. Lin, and J. Pei, "Distance-based Representative Skyline," in *ICDE*, 2009.
[10] T. Xia, D. Zhang, and Y. Tao, "On Skylining with Flexible Dominance Relation," in *ICDE*, 2008.
[11] M. L. Yiu and N. Mamoulis, "Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data," in *VLDB*, 2007.
[12] B. Reinwald and H. Pirahesh, "SQL Open Heterogeneous Data Access," in *SIGMOD*, 1998.
[13] B. Reinwald, H. Pirahesh, G. Krishnamoorthy, G. Lapis, B. T. Tran, and S. Vora, "Heterogeneous Query Processing through SQL Table Functions," in *ICDE*, 1999.
[14] M. J. Carey and D. Kossmann, "On saying "Enough Already!" in SQL," in *SIGMOD*, 1997.
[15] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting Top-k Join Queries in Relational Databases," in *VLDB*, 2003.
[16] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 1, 2001.
[17] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Joining Ranked Inputs in Practice," in *VLDB*, 2002.
[18] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A Survey of Top-k Query Processing Techniques in Relational Database Systems," *ACM Computing Surveys*, vol. 40, no. 4, 2008.
[19] "PostgreSQL: http://www.postgresql.org."
[20] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized Search Trees for Database Systems," in *VLDB*, 1995.
[21] D. Comer, "The Ubiquitous B-Tree," *Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.
[22] A. Guttman, "R-Trees: A Dynamic Index Structure For Spatial Searching," in *SIGMOD*, 1984.
[23] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *ICDE*, 2003.
[24] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries," in *VLDB*, 2002.
[25] K. C.-C. Chang and S. Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD*, 2002.
[26] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid, "Rank-aware query optimization," in *SIGMOD*, 2004.
[27] R. Agrawal and E. L. Wimmers, "A Framework for Expressing and Combining Preferences," in *SIGMOD*, 2000.
[28] J. Chomicki, "Querying with Intrinsic Preferences," in *EDBT*, 2002.
[29] ——, "Preference Formulas in Relational Queries," *TODS*, vol. 28, no. 4, pp. 427–466, 2003.
[30] W. Kießling, "Foundations of Preferences in Database Systems," in *VLDB*, 2002.
[31] M. Lacroix and P. Lavency, "Preferences: Putting More Knowledge into Queries," in *VLDB*, 1987.
[32] G. Koutrika and Y. Ioannidis, "Constrained Optimalities in Query Personalization," in *SIGMOD*, 2005.
[33] G. Koutrika and Y. E. Ioannidis, "Personalized Queries under a Generalized Preference Model," in *ICDE*, 2005.
[34] W. Kießling and G. Köstler, "Preference SQL: Design, Implementation, Experiences," in *VLDB*, 2002.
[35] R. Agrawal, R. Rantzau, and E. Terzi, "Context-sensitive ranking," in *SIGMOD*, 2006.
[36] K. Stefanidis and E. Pitoura, "Fast Contextual Preference Scoring of Database Tuples," in *EDBT*, 2008.
[37] K. Stefanidis, E. Pitoura, and P. Vassiliadis, "Adding Context to Preferences," in *ICDE*, 2007.
[38] C.-Y. Chan, P.-K. Eng, and K.-L. Tan, "Stratified Computation of Skylines with Partially-Ordered Domains," in *SIGMOD*, 2005.
[39] W.-T. Balke, U. Güntzer, and J. X. Zheng, "Efficient Distributed Skylining for Web Information Systems," in *EDBT*, 2004.
[40] A. N. Wilshut and P. M. Apers, "Dataflow query execution in a parallel main-memory environment," *Distributed and Parallel Databases*, vol. 1, no. 1, pp. 103–128, 1993.
[41] W. Jin, M. Ester, Z. Hu, and J. Han, "The Multi-Relational Skyline Operator," in *ICDE*, 2007.