Flexible and Extensible Preference Evaluation in Database Systems

JUSTIN J. LEVANDOSKI, Microsoft Research
AHMED ELDAWY, University of Minnesota
MOHAMED F. MOKBEL, University of Minnesota
MOHAMED E. KHALEFA, Alexandria University, Egypt

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 since *each* existing (and future) preference method requires implementation within the database engine. 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. This paper also provides a query optimization framework for FlexPref, as well as a theoretical framework that defines the properties a preference method must exhibit to be implemented in FlexPref. To demonstrate the extensibility of FlexPref, this paper also provide case studies detailing the implementation of seven state-of-the-art preference evaluation methods within FlexPref. We also experimentally study the strengths and weaknesses of an implementation of FlexPref in PostgreSQL over a range of single-table and multi-table preference queries.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Preference query processing, extensibility

ACM Reference Format:
Justin J. Levandoski, Ahmed Eldawy, Mohamed F. Mokbel, Mohamed E. Khalefa, 2013. Flexible and Extensible Preference Evaluation in Database Systems. *ACM Trans. Datab. Syst.* 0, 0, Article 0 (2013), 42 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. 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 [Koutrika and Ioannidis 2004]. Preference queries give users interesting an-

Authors' address: Justin Levandoski, Microsoft Research, Redmond, WA 98052; email: justin.levandoski@microsoft.com. Ahmed Eldawy and Mohamed Mokbel, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, USA; email: {eldawy,mokbel}@cs.umn.edu. Mohamed Khalefa, Alexandria University, Egypt; email: khalefa@cs.umn.edu.
This research was supported in part by the National Science Foundation under Grants IIS-0952977 and IIS-1218168.

DOI:http://dx.doi.org/10.1145/0000000.0000000

swers 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 [Chaudhuri and Gravano 1999], skylines [Börzsönyi et al. 2001], hybrid multi-object methods [Balke and Güntzer 2004], k-dominance [Chan et al. 2006a], k-frequency [Chan et al. 2006b], ranked skylines [Lee et al. 2009], k-representative dominance [Lin et al. 2007], distance-based dominance [Tao et al. 2009], ϵ-skylines [Xia et al. 2008], and top-k dominance [Yiu and Mamoulis 2007]. 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 in existence (with more on the way), 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.

The most common approach for preference evaluation in database systems is the *on-top* approach where the preference method is implemented as either a stand-alone program or a 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 [Balke and Güntzer 2004; Chan et al. 2006a; 2006b; Lee et al. 2009; Lin et al. 2007; Tao et al. 2009; Xia et al. 2008; Yiu and Mamoulis 2007]). 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 in most cases [Reinwald and Pirahesh 1998; Reinwald et al. 1999]. 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 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 [Carey and Kossmann 1997; Chaudhuri and Gravano 1999], join queries [Ilyas et al. 2003], and sorted list access [Fagin et al. 2001; Ilyas et al. 2002]. However, it is not practical to develop and maintain a database system that implements *each* existing (and future) preference method in this manner. For instance, given the amount of effort needed to implement *top-k* operations in a database system [Ilyas et al. 2008], it would be hard to replicate this effort for numerous other preference methods. Supporting each distinct preference method in this manner is simply infeasible.

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 engine, enabling the efficient execution of preference queries involving common database operations.

As depicted in Figure 1, FlexPref is implemented inside the PostgreSQL [PostgreSQL] query processor, and is extensible to arbitrary preference methods. FlexPref

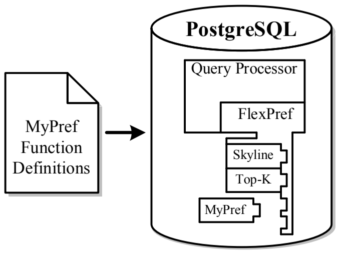


Fig. 1. FlexPref Architecture

consists of a set of generic relational operators that implement the query processing steps common to many preference methods. The generic operators themselves do not evaluate preference semantics; these semantics are injected into the FlexPref operators through the implementation of only three functions (outside of the database) 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. These functions simply define the semantics of a preference method, not how to process the preference query.

The extensible approach used by FlexPref is quite powerful: it allows a generic set of relational operators to take on the semantics of several different preference methods. In other words, FlexPref allows a database to efficiently evaluate several classes of preference queries (e.g., top-k, skyline) using the same code base. Since adding a preference method to FlexPref is quite easy (using the extensible functions), it leads to much less engineering overhead than the *built-in* approach (creating new operators for each preference method). 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 execution of preference methods inside the database engine, similar to that of the *built-in* approach. The main idea of FlexPref is to provide a set of generic, extensible operators (e.g., single-table access, join) capable of integration and optimization with existing relational operators in pipelined query plans. Then, any preference method registered with FlexPref is seamlessly integrated with the FlexPref framework, that is in turn coupled with the database 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 [Hellerstein et al. 1995]) to that of specialized indexes (e.g., B-tree [Comer 1979], R-Tree [Guttman 1984]).

We demonstrate the functionality of FlexPref through three database operations, namely, *single table access* (i.e, selection), *joins*, and *sorted list access*, that are designed to handle arbitrary preference methods integrated in FlexPref. We provide query optimization properties for all generic FlexPref operators, focusing on cardinality estimation, cost estimation, and equivalence and commutation rules for coupling FlexPref with existing relational operators. Also, we provide a theoretical framework for FlexPref that defines the properties a preference method needs to fulfill in order to be supported by FlexPref. To showcase the flexibility of FlexPref, we provide case studies for integrating seven non-trivial, state-of-the-art preference methods within FlexPref, namely, *Skyline* [Börzsönyi et al. 2001], *Top-k* [Chaudhuri and Gravano 1999], *Top-k dominating* [Yiu and Mamoulis 2007], *K-dominance* [Chan et al. 2006a], *K-frequency* [Chan et al. 2006b], ϵ*-dominance* [Xia et al. 2008], and k*-representative skyline* [Lin et al. 2007].

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 uncertain data processing and indexing. The idea is that any new functionality is implemented *only once* by the FlexPref framework, instead of re-implementing it for *each* preference method. We experimentally evaluate the strengths and weaknesses of FlexPref, implemented in the query processing engine of PostgreSQL, through the implementation several preference methods. We test our three main FlexPref operators in comparison to the *on-top* and *built-in approach*. In addition, we also experimentally evaluate the benefits of the FlexPref query optimization techniques.

The rest of this paper is organized as follows. Section 2 covers related work. Section 3 describes the usage of FlexPref. The FlexPref generic functions are described in Section 4. Section 5 covers preference evaluation in FlexPref through three main database operations. Query optimization of FlexPref operators is covered in Section 6, while Section 7 presents a theoretical framework for FlexPref by discussing supported preference method properties. Seven implementation case studies for FlexPref are discussed in Section 8. Experimental evaluation of FlexPref is provided in Section 9 while Section 10 concludes this paper.

2. RELATED WORK

2.1. Extensible Database Systems

Research in extensible relational database systems started more than two decades ago [Batory and Mannino 1986; Carey and Haas 1990] spanning academic system prototypes (e.g., EXODUS [Carey and DeWitt 1987; Carey et al. 1991], Postgres [Stonebraker et al. 1987; Stonebraker and Rowe 1986], GENESIS [Batory et al. 1988]), and commercial products (e.g., IBM Starburst [Lohman et al. 1991], Sybase [Olson et al. 1998], Oracle [Srinivasan et al. 2000]). Based on the extensible database components, previous work can be categorized into: (1) Extensibility in abstract data types [Linnemann et al. 1988; Ong et al. 1984; Osborn and Heaven 1986; Stonebraker 1986] where users can define new data types by specifying name, space allocation, and a set of functions to operate on the new data types, (2) Extensibility in query processing and optimization [Batory 1986; Graefe 1994; Graefe and DeWitt 1987; Haas et al. 1989; Kabra and DeWitt 1999; Pirahesh et al. 1992; Waas and Hellerstein 2009] where the idea is to use an extensible rule-based query optimizer to add user-defined rules, and (3) Extensibility in access methods [Hellerstein et al. 1995; Lynch and Stonebraker 1988; Srinivasan et al. 2000] where the idea is to generalize the execution of several index structures within one core implementation but allow extensible behavior based on the indexed data type, e.g., data-specific node splitting and merging strategies. Compared to this previous work, our end goal in creating FlexPref is different in two main respects. (a) FlexPref focuses *specifically* on extending the database to handle different preference methods, as opposed to focusing on generic extensibility. (b) FlexPref is a set of generic *database operators*, as opposed to a generic extensible query processor, built to abstract the common operations in preference query processing.

2.2. Preference Methods

Many methods have been proposed for evaluating user preferences over relational data. The two methods receiving the most attention are skyline [Börzsönyi et al. 2001; Chomicki et al. 2003; Kossmann et al. 2002] and top-k [Chang and Hwang 2002; Chaudhuri and Gravano 1999; Ilyas et al. 2003; Ilyas et al. 2004]. 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 [Balke and Güntzer 2004], k-dominance [Chan et al. 2006a], k-frequency [Chan et al. 2006b], ranked skylines [Lee et al. 2009], k-representative dominance [Lin et al. 2007], distance-based dominance [Tao et al. 2009], ϵ-skylines [Xia et al. 2008], and top-k dominance [Yiu and Mamoulis 2007]. 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.

2.3. 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 [Agrawal and Wimmers 2000; Chomicki 2002; 2003; Kießling 2002; Koutrika and Ioannidis 2004; Lacroix and Lavency 1987]. In some cases, the model provides rules that define how the model translates into traditional SQL queries. For example, query personalization [Koutrika and Ioannidis 2004; 2005a; 2005b] 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.

PreferenceSQL [Kießling 2002; Kießling and Köstler 2002; Kießling et al. 2011] is an extension to standard SQL supporting the best-matches only (BMO) query model. PreferenceSQL supports a number of preference operators including Pareto, Prioritization, Rank, and Dual, which can be combined to support both qualitative and quantitative preferences in a single query. The system is implemented as a middleware layer for easy integration with most database systems and supports query optimization (preference algebraic transformations, cost-based selection) as well as high-level preferences on spatial objects [Wenzel et al. 2012]. FlexPref supports a both qualitative and quantitative preference methods, as well as a hybrid of both. For example, FlexPref supports skyline (qualitative), top-k (quantitative), and top-k domination [Yiu and Mamoulis 2007] (hybrid). In PreferenceSQL terminology, the FlexPref operators support Pareto (e.g., skyline) and Rank (e.g., top-k) operations, which is a subset of those supported by PreferenceSQL since FlexPref is not closed under SV-semantics [Kießling 2002]. FlexPref differs from PreferenceSQL in that its purpose is to explore novel systems problems behind embedding generic and extensible preference operators *inside* a the core database engine.

Other work [Arvanitis and Koutrika 2012] explores embedding preferences in a relational database by extending a relation with *score* and *confidence* attributes (called p-relations). This framework defines a query processing operator that evaluates preferences according to the p-relational model, and also extends traditional operators (select, project) with p-relation semantics. Other work has explored modeling *contextual* preferences, where the objective is to evaluate preferences that change based on a user's situation [Agrawal et al. 2006; Stefanidis and Pitoura 2008; Stefanidis et al. 2007].

2.4. Preference Method Implementation

In terms of preference method *implementation*, many proposed algorithms are not designed to integrate with ad-hoc relational queries involving joins, aggregation, etc [Balke and Güntzer 2004; Chan et al. 2006a; 2006b; Lee et al. 2009; Lin et al. 2007; Tao et al. 2009; Xia et al. 2008; Yiu and Mamoulis 2007]. 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 [Chaudhuri and Gravano 1999] and join queries [Ilyas et al. 2003], and integrating skyline with join queries [Jin et al. 2007; Jin et al. 2010; Raghavan and Rundensteiner 2010] where the state of the art approach supports progressive results and early termination [Vlachou et al. 2011]. Conversely, we do not study *custom* implementations. FlexPref aims to support *any* preference method inside the database engine in a general, extensible manner. FlexPref is completely novel in this regard.

2.5. Extensible Preference Evaluation in Databases

Previous work [Levandoski et al. 2010b; 2010a] explored the creation of an extensible preference query processing framework for database systems. However, this work provided no theoretical underpinnings for the extensible framework. Furthermore, the work did not explore query optimization properties necessary for implementation in a database system. This paper expands on this previous work in the following dimensions: (1) We introduce a query optimization framework for extensible preference query processing (Section 6). Specifically, we discuss cardinality estimation for generic preference operators, cost estimation for each generic operator, and finally introduce equivalence rules for commuting generic preference operators with the select operator, distributing over joins, and commuting with projection. (2) We introduce a framework that defines the theoretical properties that a preference method must fulfill to be supported by a generic and extensible preference query processing framework (Section 7). (3) We offer case studies discussing the implementation of two state-of-the-art preference methods (ϵ-dominance [Xia et al. 2008] and and k-representative skyline [Lin et al. 2007]) that were not present in previous work. (4) We expand upon existing case studies introduced in previous work by discussing how each method supports the new query optimization properties introduced in Section 6. (5) We provide new experimental evaluation (Section 9.4) that studies the performance effects of the new query optimization framework.

3. 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.

3.1. 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 4. 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 implementations. In our system, these functions are implemented in C for easy compilation into the PostgreSQL engine (implemented in C). `DefinePreference` compiles the preference code into our framework. This process is depicted in Figure 1 for a preference method "MyPref".

3.2. 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 seven case studies of preference methods.

3.2.1. 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. Since FlexPref is implemented within the Postgres database, it supports the SQL 2008 standard, including Postgres-specific features such as extensible functions and GIS.

By default, the FlexPref is the topmost operator in a preference query. However, Section 6 discusses relational optimization of the FlexPref framework. Such optimization is non-trivial, since the optimization depend on the semantics of the preference method executing within FlexPref. In this paper, we study and experiment with FlexPref integrated alongside the select, project, and join operations. Integrating FlexPref with aggregation and group-by operations is a property of the specific preference method implemented within FlexPref, and is a topic of future work.

3.2.2. Seven Query Examples. Using the query skeleton of FlexPref, we now give use case examples for seven state-of-the-art preference methods, namely, *skyline* [Börzsönyi et al. 2001], *top-k* [Chaudhuri and Gravano 1999], *top-k dominating* [Yiu and Mamoulis 2007], *k-dominance* [Chan et al. 2006a], and *k-frequency* [Chan et al. 2006b]. 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, not 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 $d3$.

Case Study VI: k-Representative Skyline The k-representative skyline [Lin et al. 2007] is based on the same dominance property of the traditional skyline method. However, this method scores each skyline answer by the number of other objects it dominates, and returns the k objects with the highest score. The `using` clause for the k-representative skyline method is:

```
    Using K-Rep-Skyline With K=3 Objectives MIN d1, MIN d2, MAX d3;
```

In this clause, K=3 answers are required, while the preference objectives are exactly the same as the skyline case previously discussed.

Case Study VII: Epsilon Dominance The epsilon-dominance (abbr. ϵ-dominance) preference method [Xia et al. 2008] alters the concept of traditional skyline dominance to be more flexible. The idea is to increase or decrease the dominance region of each object in the dataset by a constant ϵ (details covered in Section 8). The `using` clause for the ϵ-dominance is:

```
    Using Epsilon-Dominance With E=1.5 Objectives MIN d1, MIN d2, MAX d3;
```

In this clause, E=1.5 denotes that the dominance region should be increased by a factor of 1.5, while the preference objectives are to minimizing both the price and distance attributes, while maximizing the rating.

4. 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 3.1. These functions are used by the generic FlexPref operators during query processing, which we describe in Section 5. We also provide seven case studies for how these functions can be used to implement various preference methods in Section 8.

Before discussing the three functions, we describe two macros that will be used by our functions and the query processing techniques in Section 5

— `#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 cannot become a preferred object, 0 otherwise. Alternatively, return -2 if the preference method does *not* rely on pairwise comparison; returning -2 in this case leads to optimizations in the preference evaluation algorithm.
— `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 database query processor internals. 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`). FlexPref also provides a third function, `AddPreferredToSet`, to allow the preference method to maintain the order of objects and cardinality of its running set of preference answers. 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 practice, implementing each of these functions is quite simple. We implemented seven state-of-the-art preference methods in FlexPref; none of the functions we implemented exceeded fifteen lines of code (details in Sections 8). In later sections, we discuss additional functions necessary to provide further query processing optimizations. Section 5.3 discusses an additional function to optimize FlexPref for sorted data. Meanwhile, Section 6 discusses functions to allow a preference method implemented in FlexPref to optimize with existing relational operators.

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. We defer a detailed discussion of preference method support to Section 7, where we explore a taxonomy of theoretical properties that preference methods must meet in order to be supported by FlexPref.

5. PREFERENCE EVALUATION IN FLEXPREF

This section explores the details of preference evaluation in FlexPref that uses the three main functions, `PairwiseCompare`, `IsPreferredObject`, and `AddpreferredToSet`, de-

Algorithm 1 Single Table Access in FlexPref

```
 1: Function SingleTableAccess(TableReference $T$)
 2: Preference Set $S \leftarrow \Phi$
 3: for each Object $P$ in $T$ do
 4:     $P_{score} \leftarrow$ DefaultScore
 5:     for each Object $Q$ in $T$ do
 6:         cmp $\leftarrow$ PairwiseCompare($P$,$Q$)
 7:         if cmp = 1 then
 8:             if $Q \in S$ 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:         if cmp = -2 then exit inner loop (go to line 17)
16:     end for
17:     if IsPreferredObject($P$,$S$) then AddPreferredToSet($P$,$S$)
18: end for
19: return $S$
```

scribed in Section 4. We will first present single table access, i.e., selection queries over single table, in FlexPref in Section 5.1. Then, in Section 5.2 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 5.3. 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 [Chan et al. 2005]).

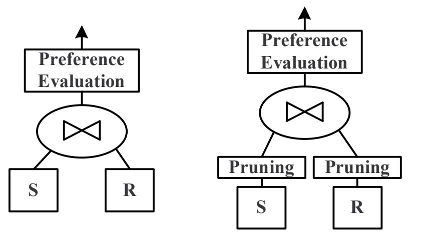
5.1. Single-Table Access

Single table access selects a set of preferred objects from a single table. For instance, all the query examples given in Section 3 use single table access where the objective is to retrieve the set of preferred restaurants according to the given preference criteria, where all data is stored in a single table R. We propose a block-nested loop (BNL) algorithm to execute single-table preference evaluation. We chose a BNL approach for two main reasons: (1) it is simple and appropriate for a generic framework since it is known to work for executing a number of diverse preference methods; (2) it is appropriate for cases when data is *not* indexed, which is a common case that must be handled by a database. We discuss further optimizations for index access in Section 5.3.

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 4. While simple, this single execution framework is very powerful as it can accommodate many different preference evaluation methods. To illustrate, Section 8 covers the implementation of seven 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.

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



(a) Naive join (b) FlexPref join

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

(c) Example data

Fig. 2. Join operator

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. If `PairwiseCompare` returns -2 (i.e., the preference method does not rely on pairwise comparison of objects), the algorithm breaks out of the inner loop. 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 8, 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.

5.2. Multi-Table Access

The join operation is one of the most common, and expensive, operations in a DBMS. Joins are also an integral part of preference queries as well. For example, consider a 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. 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 special indexing structure. In fact, the approach is applicable to any join method (e.g., hash join, index-nested loop). 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.

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 2(c). 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 (similar pruning concepts have been proposed in [Chomicki 2003; Hafenrichter and Kießling 2005; Endres and Kießling 2011]).

Pruning in Algorithm 2 works for the following reason. For 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 "better" than tuples in N within 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 worse than those in P, then tuples in N cannot become 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.

To make the pruning concept concrete, we now provide a brief example for both the skyline and top-k preference methods (examples for more methods can be found in Section 8). Consider generating the skyline by joining tables R and S in Figure 2(c) using the predicate R.id=S.id. Focusing on table S, we can safely prune four records {*(a,3,4),(a,4,3)*} and {*(b,4,2),(b,3,6)*} since they are dominated within their join key groups by *(a,5,5)* and *(b,8,8)*, respectively. These pruned records have no chance of being skylines when joining with a matching tuple from table R (*(a,5,3)* and *(b,7,2)*, respectively). Similarly, consider generating a top-k answer by joining R and S (R.id=S.id), where $k = 2$ and we use a monotonic function *F(d1,d2,d3,d4)*=$\sum(d_i)$ to score each record. In this case we can safely prune *(b,4,2)* and *(a,3,4)* (or equivalently *(a,4,3)*) from S, since neither are the top-2 records within their join key groups according to *F(d3,d4)*. Likewise, since the pruned tuples were not top-2 within their own join key group, they will not contribute to the overall top-2 answer if they were to join with a matching tuple from R.

5.3. Sorted List Access

This section explores how to efficiently perform preference evaluation if each of the attributes in the `PREFERRING` clause is available in sorted order. For instance, each attribute could be stored in a 2-ary table as (id, attribute value) tuples (i.e, a fully decomposed storage model [Copeland and Khoshafian 1985]), or the attributes could be

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 */
```

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

Fig. 3. Sorted list example

indexed by a B-tree. The main idea we explore here is to generate a complete and correct preference answer after reading only a *fraction* of the sorted data. This approach could potentially reduce the I/O overhead compared to query processing over unsorted or non-indexed data.

Figure 3 gives an example of three attributes stored using the decomposed storage model, where each table: (a) includes the tuple ID, and (b) is sorted on the attribute value. 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* [Ilyas et al. 2003] and *skyline* [Balke et al. 2004]. This section presents a generic algorithm that exploits sorted lists for query efficiency that works for *any* preference method compatible 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, say we read the first tuple from each list in Figure 3. 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 4.

— `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` are named virtual objects since they store the last and first values read from each input list, respectively. For example, reading round-robin twice from each list $D1$ to $D3$ in Figure 3 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

Algorithm 3 Sorted List Access in FlexPref

```
1: Function GeneralSortedAccess(Lists[$n$], $CheckInterval$)
2: $stop \leftarrow$ false; $count \leftarrow 0$
3: Partial Set $P \leftarrow \Theta$
4: $\forall_{i<n}$ O[i]=$\Theta$; F[i]=$\Theta$
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 $count(modulo)CheckInterval = 0$ then
11:        $stop \leftarrow$ StopSortedEval($P$, $O$, $F$) ;
12:    else
13:        increment $count$
14:    end if
15: end while
16: for each incomplete point $q \in P$ do
17:     $\forall_j$ 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 */
```

assume tuples are combined using $t.id$. The algorithm also takes as input an integer $CheckInterval$, used to throttle the check for a stopping condition after the number of round-robin list retrievals specified by $CheckInterval$ value. Calling `StopSortedEval` after every attribute retrieval may be expensive, so the CheckInterval value is implemented for efficiency reasons. It has no effect on the correctness of the algorithm, since it is alright to check for the stopping condition less often. 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 this iteration must check for stopping condition (i.e., count module $CheckInterval$ equals zero), the boolean *stop* is set by calling extensible function `StopSortedEval`. 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 [Wilshut and Apers 1993] between n 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.

In addition, it is possible to use the sorted-list algorithm within the pruning step of the preference join algorithm (Algorithm 2). We can simply replace the calls to *SingleTableAccess* in lines 2 and 3 of Algorithm 2 with calls to *GeneralSortedAccess*. Of course, this assumes that all attributes in either the left or right join relation are available in sorted order (e.g., indexed by a B-tree).

6. QUERY OPTIMIZATION WITH FLEXPREF

Query optimization is responsible for the selection of efficient pipelined query execution plans. In some cases, the difference between a good and bad query plan can result in an order of magnitude performance difference. In this section, we explore query optimization techniques for the generic FlexPref operators necessary to integration with a traditional relational query optimizer. The framework discussed in this section allows for FlexPref to become a first-class citizen in a database engine, and allows the query optimizer to consider efficient query plans that contain generic FlexPref operators coupled with traditional relational operators (i.e., select, project, join). We first explore cost estimation of FlexPref operators under difference scenarios. Next, we explore FlexPref support for cardinality estimation. Finally, we explore a number of relational algebra equivalence rules for FlexPref, studying commutability with the following standard relational operators: selection, join, and projection.

Since FlexPref is a generic framework, many of the query optimization rules depend on the semantics of the preference method executing within FlexPref. Many of these rules are known [Chomicki 2003; Endres and Kießling 2011; Hafenrichter and Kießling 2005]. Our goal in this section is not to develop new query optimization rules for each preference method that could be implemented within FlexPref. Rather, we discuss how query optimization may be achieved in relational database system that implements the FlexPref engine. In many cases, we revisit existing query optimization techniques, and classify which techniques apply to FlexPref.

6.1. Cardinality Estimation

The output cardinality of the FlexPref operator (both single table and join) depends completely on the semantics of the preference method executing within FlexPref. For example, for preference methods capable of producing k results (e.g., top-k, k-frequency, top-k domination), the cardinality when implemented in the FlexPref operator is obviously k. However, the answer to a skyline query is not a total order, thus the answers cannot be "ranked" to produce exactly k results. Therefore, several skyline cardinality estimation techniques have been developed based on statistical sampling of the input relations [Chaudhuri et al. 2006; Zhang et al. 2009].

Since output cardinalities are unique to a preference method, FlexPref provides a plugin function to provide the query optimizer with the output cardinality of a specific preference method executing within FlexPref. The signature for this function is as follows.

— `EstimateCardinality(InputCard C, [Optional] InputRelation R)`: Given the input cardinality, and optionally a reference to the input relation (if available), return the expected cardinality of the preference method implemented within the FlexPref framework.

The input to this function is (1) the cardinality of the input data. This value can be the cardinality of a database table, or the estimated cardinality derived by the query optimizer for an operator that feeds data to the FlexPref operator. (2) If the input to FlexPref is a relational table R (and not a pipeline operator), a reference to R is passed to the estimation function. This reference is provided in order to accommodate techniques that benefit from sampling the underlying relation (e.g., skylines [Chaudhuri et al. 2006]). The *EstimateCardinality* function is invoked as part of the background statistics collection and caching process common in most database systems. The function does not need to be invoked after each query. Rather, the database may call the function once to retrieve and cache initial cardinality statistics, and call it again in the future if it suspects statistics have changed significantly.

6.2. Cost Estimation

In this section, we provide an analysis of the cost estimates for the three versions of the FlexPref operator presented in Sections 5.1 through 5.3. Our cost metric is page I/Os: the most common database cost metric.

6.2.1. Single-table FlexPref Operator. The single-table FlexPref is implemented as a block-nested loop algorithm (i.e., Algorithm 1). We assume the input relation T is stored on N pages, and that B pages are available in each block.

Two factors of the preference method influence the cost estimate of the FlexPref single-table operator: (1) whether the preference method requires exhaustive pairwise comparison (i.e., the plugin function `PairwiseCompare` does *not* return -2), and (2) whether the preference method is transitive. We identify three cases where the combination of these factors leads to different cost estimates.

Case 1. If the preference method requires exhaustive pairwise comparison and is not transitive, the cost can be estimated as $N(N-B)$. Essentially, N pages will be read in the outside loop of Algorithm 1, B pages at a time. The number of pages read in the inside loop is $(N-B)$, which is the rest of the pages in relation T less the B pages already in memory (read for the outside loop).

Case 2. If the preference method does not require exhaustive pairwise comparison, regardless of whether it is transitive, the cost is estimated as N (i.e., a single scan). Essentially, if a score can be derived for an object without comparing it to each other object in T, the inside loop of Algorithm 1 is not needed.

Case 3. If the preference method is transitive and requires pairwise comparison, the cost is estimated as $(N-P)(N-(B+P))$. The variable P represents the amortized number of pages that can be discarded during runtime due to preference method's transitivity property. In this case, if enough tuples are discarded from in-memory pages during preference evaluation, these pages can be compacted, meaning less page reads in subsequent passes of the nested-loop algorithm.

6.2.2. Multiple-table FlexPref Operator. The cost of the multi-table FlexPref operator consists of the sum of three costs: *pruning*, *joining*, and the final on-top *preference evaluation* (Section 5.2). Recall that the *pruning* step consists of applying the single-table FlexPref operator to each join key group for both join input tables (Algorithm 2). We focus on the cost of pruning a single join input table, as the analysis is symmetric for both inputs. Given a join input consisting of N pages, there are two cases that yield different I/O costs: (1) If the join key group is clustered, the cost of pruning is N I/Os, as pruning can be done in a single pass. This cost assumes the number of pages storing each join key group fit in memory, which is very likely. (2) Otherwise, the cost of pruning a join input is the I/O cost of performing a *group by*, necessary to group the join key attributes. We assume the pruning step is performed on in-memory pages resulting from the *group by*.

To predict the cost of the *join* operation, we can use any existing cost function for any standard database join (e.g., [Shapiro 1986]), since FlexPref functions regardless of the join method used. However, the cost function employed will be reliant on the input cardinalities in order to yield an accurate estimate. Since the pruning step removes tuples from the join input, we must find a reliable estimate for the join input cardinalities. Given input relations R and S of size N and M pages, respectively, the size (in pages) of the inputs to the join is (N-P_R) and (M-P_S), where P_R and P_S represent the number of join input pages saved by the pruning operation over R and S, respectively. The size of P_R can be estimated by applying the `EstimateCardinality` function (described in the previous section) to the average size of each join key group for table R in order to yield an estimated prune cardinality. Subtracting this prune cardinality from the cardinality of R yields the number of pruned tuples; dividing this number by the database page size yields the value of P_R. Calculating P_S is similar.

The cost of the final *preference evaluation* is the cost of the single-table FlexPref operation performed after the join. This final cost estimate is necessary, since the final preference evaluation is non-blocking, thus cannot be performed on the fly. This cost was covered in Section 6.2.1, with the exception that the input size is number of pages produced by the join operation.

6.2.3. Sorted List FlexPref Operator. The I/O performance of the FlexPref sorted list operator depends heavily on the stopping condition (implemented in plugin function *StopSortedEval*), and the correlation of each of the D attributes in the preference query (e.g., anti-correlated, correlated, or independent). Modeling an *exact* cost function based on these properties is extensive, and outside the scope of this paper. However, the average cost of the sorted list operator is $O(N)$, derived as follows. Given D attributes in a preference query, the best case cost is $D * 1$, meaning a single page is read from each input relation. Since D is a constant, the best case is $O(1)$. The worse case cost is $D * N$, i.e., $O(N)$ since all pages must be read from each sorted relation. The average case cost is $\frac{DN}{2}$, where only half the data is read from each sorted relation.

6.2.4. Cost of Plugin Functions. Given the cost analysis done so far, the primary I/O overhead for each generic FlexPref operator is dominated by its generic data retrieval steps. The custom plugin functions for each operator are designed to require only simple and straightforward operations (e.g., tuple comparison, list insertion) that we believe lead to negligible I/O overhead. In the rest of this section, we justify this claim, referring to the non-plugin steps executed by FlexPref as *primitive* operations, and the extensible steps as *plugin* operations.

Based on our previous analysis and our experience implementing existing preference methods in FlexPref (see Section 8), we believe the plugin operation overhead will be small compared to the rest of the query processing framework. For instance, consider the generic functions *IsPreferredObject* and *AddPreferredToSet*. For the single-table FlexPref operator (Algorithm 1), these functions have the potential to incur the most I/O, since their plugin operations are responsible for maintaining and organizing data (tuples) in the current preference answer. For preference methods that report k answers (e.g., top-k, k-frequency), the maximum size of the maintained preference answer will be k. Since k is likely to be small (compared to the data set size), the plugin operations will operate on data (i.e., the answer set) that fits on (at most) a few memory-bound pages, leading to negligible (if any) I/O costs. Meanwhile, preference methods that *cannot* maintain an answer set based on total rank order are less predictable. Focusing on skylines, given a data set with d independent dimensions and a cardinality of n, the expected number of skyline objects is $\Theta(ln^{d-1}n/(d-1)!)$ [Godfrey et al. 2005]. However, the skyline size can be considerably larger in the worst case [Chaudhuri et al. 2006; Godfrey 2004].

Special consideration is required for the sorted list plugin operations of function *StopSortedEval*. This function operates over a potentially large data structure containing partial tuples. Since the cardinality of the partial tuple data could reach the size of the input data, the *StopSortedEval* function has the potential to incur non-trivial I/O in this worse case. However, since the call to *StopSortedEval* can be reduced to every M^{th} iteration of the sorted list FlexPref algorithm, the I/O overhead of *StopSortedEval* is likely to be amortized over the runtime of the sorted list algorithm. Thus, the I/O cost of the sorted list algorithm will still be dominated by the primitive FlexPref operations.

6.3. Equivalence Rules

Along with cost and cardinality optimization, the last piece of information needed to optimize preference queries with FlexPref is a set of algebraic equivalence rules. In this section, we present relational algebra equivalences necessary for the database to optimize FlexPref in a standard select-project-join (SPJ) query. We first discuss commuting selection with the FlexPref operator. We then discuss distributing FlexPref over the join operations. We note that the join distribution rule studied serves a different purpose than the multi-table algorithm presented in Section 5.2. The purpose of join distribution is to study an equivalence that allows us to "push down" the single-table FlexPref operator before a standard join operation, without having to perform preference evaluation again *on top* of the join. Finally, we present rules for commuting FlexPref with projection.

In the rest of this section, we symbolize the FlexPref operator with the algebraic notation $\mathcal{F}_{\Theta\mathcal{P}}$. The symbol Θ represents the preference method implemented in FlexPref (e.g., Θ = skyline). The symbol $\mathcal{P}$ represents the set of preferences specified in our extended SQL clause `Objectives` presented in Section 3.2.1. For instance, preference objectives $\mathcal{P}$ for skyline could be `min price` and `min distance`.

6.3.1. Commuting Selection with FlexPref. The ability to commute selection with FlexPref is one of the most important optimizations necessary to efficient preference query processing. Due to the cost of the FlexPref operator in the average case (covered in Section 6.2.1), pushing selection can greatly increase query efficiency by data input into FlexPref. Formally, we study the following equivalence.

$$\sigma_{\mathcal{C}}(\mathcal{F}_{\Theta\mathcal{P}}(R)) = \mathcal{F}_{\Theta\mathcal{P}}(\sigma_{\mathcal{C}}(R))$$

Where $\mathcal{C}$ is the selection condition, and R represents a single relational input.

Unfortunately, due to the wide variety of preference methods that can be implemented within the FlexPref framework, there is no standard selection commutability law that applies to FlexPref. In fact, a preference method can fall into one of three classifications for commutability with the selection operation.

(1) *Always commutes*. In this class, the semantics of the preference method allow selection to be placed before or after preference evaluation, regardless of the selection condition. A prime example of a preference method in this class is ranking [Li et al. 2005]. Since the rank value of each object is only a function of the attributes of the object (i.e., the rank does not require comparison to other objects), ranking commutes with selection regardless of the preference objectives $\mathcal{P}$ or selection condition $\mathcal{C}$.

(2) *Conditionally commutes* In this class, selection commutability is conditional upon the constraints in either the selection predicate $\mathcal{C}$, the preference objectives $\mathcal{P}$, or a combination of both. An example of a preference method in this class is the skyline [Börzsönyi et al. 2001] method, where it has been shown that, in order to commute (a) the selection condition must be specified over an attribute that is also a preference objective and (b) the selection condition cannot contradict the preference objective [Chomicki 2003].

(3) *Never commutes* In this class, the semantics of the preference method never allow commutability with selection. An example of a preference method in this class is top-k dominance [Yiu and Mamoulis 2007]. Since this method ranks each object O based on the number of other objects O dominates in relation R, performing selection before preference domination can remove important tuples and thus alter rank values, changing the final preference answer.

In order to correctly optimize preference queries involving any three of these selection commutability classes, the FlexPref framework extends the query optimizer with a plugin function that specifies whether the running preference method commutes with selection. This plugin function is written by the preference method implementer.

— `SelectionCommute(Selection condition` $\mathcal{C}$`, Preference objectives` $\mathcal{P}$`)`: Give the selection condition and preference objectives, return whether the preference method can commute with the selection operation.

This function returns true for preference methods that *always commute*, and false for methods that *never commute*. Thus, implementation of this function is necessary for functions that *conditionally commute*. In Section 8, we study the selection commutability rules implemented in this function for our seven case study methods.

6.3.2. Distributing FlexPref over EquiJoins. For FlexPref to distribute over the join operation on two relations R and S, we are required to "break up" the preference objectives in $\mathcal{P}$ into: (1) $\mathcal{P}_R$, the objectives that apply to attributes in R, and (2) $\mathcal{P}_S$, the objectives that apply to attributes in S. As an example, consider a skyline query over relations *Hotels S* and *Restaurants R* with $\mathcal{P}$={min S.price, max S.rating, min R.price, max R.rating}. Here, $\mathcal{P}_R$={min R.price, max R.rating}, while $\mathcal{P}_S$={minS.price, max S.rating}. Formally, we investigate the following equivalence.

$$\mathcal{F}_{\Theta\mathcal{P}}(R \bowtie_{\mathcal{J}} S) = \mathcal{F}_{\Theta\mathcal{P}_R}(R) \bowtie_{\mathcal{J}} \mathcal{F}_{\Theta\mathcal{P}_S}(S)$$

Where $\mathcal{J}$ represents the join predicate. Note that if $\mathcal{J} = \phi$, the join is a Cartesian product.

The distribution of FlexPref over a join is dependent on three main factors: (1) the preference method Θ, (2) the join predicate $\mathcal{J}$, and (3) whether $\mathcal{P}$=$\mathcal{P}_R$ or $\mathcal{P}$=$\mathcal{P}_S$, that is, whether the preference objectives apply to attributes in only R or S. No standard join distribution law applies to FlexPref. For instance, the skyline preference method has been shown to distribute with Cartesian product [Chomicki 2003] (i.e., when $\mathcal{J} = \phi$). Skyline also distributes over a join when $\mathcal{P}$=$\mathcal{P}_R$ (or $\mathcal{P}$=$\mathcal{P}_S$) in the case of an equijoin, i.e., the preference objectives apply to attributes only in R or S. However, skyline does not distribute for an equijoin when objectives in $\mathcal{P}$ apply to attributes in *both* R and S. Meanwhile, the top-k method does not even distribute over a Cartesian product. Assuming $\mathcal{P}$ applies to attributes in both R and S, the top-k evaluation cannot be "pushed down" below the Cartesian product, as this transformation would produce k inputs to the Cartesian product from both R and S, meaning k^2 results would be generated from the Cartesian product. Essentially, this "push down" rule will not work for any method designed to produce k results (e.g., top-k domination) without also integrating preference evaluation *within* or *on-top* of the join (e.g., see [Li et al. 2005]).

In order to correctly optimize preference queries involving various preference methods exhibiting different distribution rules for join, the FlexPref framework provides a plugin function that specifies whether the running preference method commutes with a join operation. The definition of this function is as follows.

— `JoinDistribute(Preference objectives` $\mathcal{P}$`, Join predicate` $\mathcal{J}$`,` R `Attributes` A_R`,` S `Attributes` A_S`)`: Given a set of preference objectives, the join predicate, the tuple attributes of input relation R, and the tuple attributes of input relation S, return whether the preference method can distribute over the join operation.

In Section 8, we provide the implementation details of this plugin function for our seven case study methods, along with extensive examples.

6.3.3. Commuting Projection with FlexPref. Given a projection operation $\pi_{\mathcal{V}}$, where $\mathcal{V}$ is the set of projected attributes, we investigate the following equivalence.

$$\pi_{\mathcal{V}}(\mathcal{F}_{\Theta\mathcal{P}}(R)) = \mathcal{F}_{\Theta\mathcal{P}}(\pi_{\mathcal{V}}(R))$$

Table I. Binary relation properties supported by FlexPref

Property	**Definition**	**FlexPref Support**
Reflexive	$t_x \succ_p t_x, \forall t_x \in R$	Not Allowed
Irreflexive	$\neg(t_x \succ_p t_x), \forall t_x \in R$	Required
Symmetric	$(t_x \succ_p t_y) \Rightarrow (t_y \succ_p t_x)\ , \forall t_x, t_y \in R$	Not Allowed
Transitive	$(t_x \succ_p t_y) \wedge (t_y \succ_p t_z) \Rightarrow (t_x \succ_p t_z)\ , \forall t_x, t_y, t_z \in R$, $t_x \neq t_y \neq t_z$	Allowed
Asymmetric	$(t_x \succ_p t_y) \Rightarrow \neg(t_y \succ_p t_x)\ , \forall t_x, t_y \in R$	Allowed
Antisymmetric	$(t_x \succ_p t_y) \wedge (t_y \succ_p t_x) \Rightarrow (t_x = t_y)\ , \forall t_x, t_y \in R$	Allowed
Connective	$(t_x \succ_p t_y) \vee (t_y \succ_p t_x) \vee (t_x = t_y)\ , \forall t_x, t_y \in R$	Allowed

Projection will always commute with FlexPref, no matter the preference method Θ, as long as $\mathcal{P} \subseteq \mathcal{V}$, i.e., the set of attributes in the preference objectives are contained in the set of projected attributes. If this case does not hold, and projection is performed before FlexPref, at least one attribute in $\mathcal{P}$ will be discarded prior to preference evaluation, which is illegal.

7. SUPPORT FOR PREFERENCE METHOD PROPERTIES IN FLEXPREF

In this section, we present a taxonomy of theoretical properties that are supported (and not supported) by FlexPref. The goal here is to present a road map for database practitioners, such that creators of new (or existing) preference methods can easily tell if their method qualifies for implementation in FlexPref. Looked at another way, the content of this section serves as a theoretical foundation of FlexPref, as it defines the boundaries of what is theoretically possible to accomplish within the FlexPref framework. In the rest of this section, we discuss our taxonomy of theoretical properties broken down into the following five classes: (1) binary preference relations, (2) preference granularity, (3) preference definition types, (4) preference composition, and (5) deterministic and non-deterministic answers.

7.1. Binary Preference Relation

In this section, we explore seven binary relation properties that preference methods must meet for implementation in FlexPref. These properties are listed in Table I, and have become standard in categorizing preference evaluation approaches [Chomicki 2003; Koutrika et al. 2010; Stefanidis et al. 2011]. We note that these properties also have a corresponding graph representation [Koutrika et al. 2010; Stefanidis et al. 2011]. For each property, we give: (a) A formal definition, in terms of a binary preference relation $\succ_p$ defined between two tuples in a relation R. These formal definitions are given in Table I (b) A classification of whether FlexPref supports preference methods that exhibit the property. Three classifications are possible: (1) *Not Allowed*, the semantics of the property are not supported by FlexPref; (2) *Required*, the preference method must adhere to the property to be implemented in FlexPref; (3) *Allowed*, FlexPref is indifferent to whether the preference method exhibits the property.

7.1.1. Reflexive. The reflexive property states that all tuples can dominate themselves. This property makes (non-empty) preference answer derivation impossible, and is not allowed in FlexPref.

7.1.2. Irreflexive. The irreflexive property states that no tuple can dominate itself. In FlexPref, a tuple is never compared with itself in the generic query processing framework. Thus, embedded preference methods must be irreflexive, i.e., no tuple can dominate itself.

Table II. Preference granularity properties supported by FlexPref

Granularity	**Definition**	**FlexPref Support**
Tuple	Preferences defined at the tuple level, preference evaluation performed between two individual database tuples	Supported
Set	Preferences defined between a set (or cluster) of common, preference evaluation performed between tuple sets	Limited Support
Attribute	Preferences defined between attributes, used to determine which attributes are more important in preference evaluation	Not Supported

7.1.3. Symmetric. The symmetric property states that for all pair of tuples, the dominance property is symmetric (i.e., each pair can dominate each other). Preference functions exhibiting the symmetric property are not allowed in FlexPref. If all tuples can dominate each other, deriving a non-empty preference answer is impossible.

7.1.4. Transitivity. FlexPref handles both transitive and non-transitive preference methods. As discussed in Section 6.2, transitivity leads to optimizations in FlexPref query processing, and more query processing overhead for non-transitive methods.

7.1.5. Asymmetric. The asymmetric property states that for all tuple pairs t_x and t_y, if t_x dominates t_y, then t_y can never dominate t_x. FlexPref is indifferent to whether a preference method exhibits the asymmetric dominance property, as dominance is evaluated both ways for all tuple pairs. It is important to note that a non-symmetric preference method does *not* imply the method exhibits symmetry.

7.1.6. Antisymmetric. The antisymmetric property states that for all tuple pairs t_x and t_y, if the pair exhibits symmetric dominance then t_x and t_y are equivalent. FlexPref can support preference methods where tuples dominate each other but are not equivalent (discussed previously for the asymmetric case). Thus, FlexPref is indifferent to the antisymmetric property.

7.1.7. Connective. The connective property implies a total order (or strong order) can be derived using the preference method. The ability to assign a numeric score to each tuple in FlexPref implies that FlexPref supports connective preference methods (e.g., top-k). However, scoring tuples is not a requirement in FlexPref, thus non-connective methods (e.g., skylines) are also supported.

7.2. Preference Granularity

Preference granularity refers to the "level" at which preferences are expressed and evaluated. Table II summarizes the preference granularity properties supported by FlexPref. There are three granularity properties we consider.

7.2.1. Tuple-level granularity. Preferences at this granularity are expressed at the tuple level, usually through the values of their attributes (e.g., minimize price, maximize rating). Preference evaluation is performed between two individual database tuples. Tuple-level preference granularity is the most widely used in practice (e.g., skyline [Börzsönyi et al. 2001], k-dominance [Chan et al. 2006a]). FlexPref supports tuple-level preference granularity, as its core functionality compares tuples pairwise in building a preference answer.

7.2.2. Set preferences. Set properties allow users to express preferences constraints over a set of tuples. For instance, a user may prefer a set of four restaurants, minimizing price and distance, where two of them should be Chinese restaurants. FlexPref supports set preferences in a limited fashion. FlexPref *does not* support set preferences at neither the syntax nor query processing level. However, FlexPref *does* support a lim-

Table III. Preference definition properties supported by FlexPref

Property	**Definition**	**FlexPref Support**
Numeric	Preference expressions over numeric tuple attributes	Supported
Categorical	Preference expressions over categorical tuple attributes	Limited
Contextual	Numeric or categorical preference expressions valid only under given contextual constraints	Not Supported

ited form of set preferences if the set can be expressed using aggregation and transformed into a single tuple. In this case the FlexPref operators can evaluate the result tuples from the aggregate operator(s).

7.2.3. Attribute preferences. Attribute preferences allow the user to express preferences *between attributes*, such that the top-n attributes are included in the preference query [Koutrika and Ioannidis 2004]. For instance, one could specify that preferences over a restaurant price attribute should matter more in a preference query than restaurant rating. FlexPref does not support attribute-granularity preferences at the syntax nor query processing level.

7.3. Preference Definition

Preference definition properties refer to how a user expresses preferences, and what type of data a user can express preferences over in the system. We consider a classification of four different preference definition properties in FlexPref. Table III summarizes these properties and their support in FlexPref.

7.3.1. Numeric Preferences. Numeric preferences refers to support for preferences defined over numeric attributes. FlexPref supports numeric preferences using the MIN and MAX expressions. Numerical ranking (i.e., top-k processing) is supported by specifying a monotonic ranking function as a user-defined function within the `With Objectives` clause (see the top-k example in Section 3.2.2). FlexPref also supports trivial substitutable value (SV) semantics [Kießling 2002] on single attributes. An example is the AROUND preference from PreferenceSQL that substitutes a stored value x with its distance to a given value v, formally, $abs(x - v)$. FlexPref supports this class of preferences by allowing user-defined functions in its `With Objectives` clause(e.g., `MIN abs(x-v)`). These functions are applied to a tuple before processing by the FlexPref operator.

7.3.2. Categorical Preferences. The focus of the FlexPref framework is primarily on numerical attributes. However, FlexPref provides *limited* support for categorical preferences by translating categorical preferences to numerical preferences. We classify this support as limited since the translation must be performed as a pre-processing step using a set of categorical preferences given by the user.

To perform the translation, we assume that categorical preferences are expressed using the common "better-than" graph [Kießling 2002; Wong et al. 2008]. A better-than graph is a finite, acyclic, directed graph, where nodes represent values of a category attribute, and a directed edge between nodes specifies that the source node is preferred over the destination node. For example, assume a user expressed the following preferences for a restaurant type categorical attribute: (Thai$\rightarrow$ Chinese), (Italian$\rightarrow$ French), (French$\rightarrow$ Chinese). From these preferences, we can construct a better than graph as depicted in Figure 4, where each node is assigned a level value (lower is better). The level is the minimum number of graph traversals, starting from a "best" node, necessary to reach the destination node, where a "best" node is a node with no incoming edges (Italian and Thai at level 0 in Figure 4). All unspecified nodes are assigned the highest level, labeled "everything else" in Figure 4. Substituting the numeric level

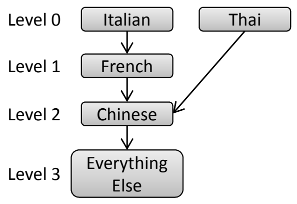


Fig. 4. Categorical preferences represented as "better-than" graph

value for each category value, we can then transform the categorical preference into numeric preference expression that minimizes the level value.

7.3.3. Contextual Preferences. Contextual preferences are preference expressions that are valid only if a given contextual constraint holds. For example, a user may prefer expensive French restaurants over Chinese restaurants only if the weather is sunny. Contextual preferences can be qualitative [Agrawal et al. 2006] or quantitative [Stefanidis and Pitoura 2008].

FlexPref supports neither type of contextual preference at the syntax or query processing level due to the high overhead needed to process contextual preference queries. In general, support for contextual preferences requires an offline "reconciliation" step used to disambiguate conflicts in contextual constraints. For instance, a user may have the following two conflicting preferences: (1) French restaurants preferred over Chinese restaurants if the weather is sunny; (2) Chinese restaurants preferred over French restaurants if weather is sunny and cold. In this example, a reconciliation step is needed to break the cyclic preference for French and Chinese restaurants. This reconciliation step is NP-hard [Agrawal et al. 2006]. There have been recently been modular approaches to constructing context-aware preferences [Roocks et al. 2012], though FlexPref does not support this approach.

Table IV. Preference composition properties supported by FlexPref

Composition	**Definition**	**FlexPref Support**
Pareto composition	Base preferences given equal importance in preference method	Supported
Prioritized composition	Base preferences can be prioritized, some more important to preference method than others	Not Supported

7.4. Preference Composition

Preference composition refers to how a preference method assumes base preferences are combined. In FlexPref, multiple base preferences are listed in the `PREFERRING` clause, separated by the `AND` keyword. Examples of two base preferences for restaurant data are *minimize price* and *maximize rating*. Two alternatives exist for preference composition; their support in FlexPref is summarized in Table IV.

7.4.1. Pareto composition. Pareto composition treats all base preferences as equally important. Due to its commonality, Pareto composition with trivial substitutable semantics [Kießling 2002] is supported by FlexPref.

7.4.2. Prioritized composition. Prioritized composition allows users to "rank" base preferences based on importance [Kießling 2002]. For instance, a user may state that minimizing *price* is more important than maximizing *rating* when looking for a restaurant. Prioritized preference composition is used very little in practice, as it is not used by

Table V. Deterministic and non-deterministic answers in FlexPref

Property	**Definition**	**FlexPref Support**
Deterministic answers	Query processing logic has ability to produce deterministic answer	Supported
Nondeterministic answers	Query processing logic has ability to produce nondeterministic answer	Supported

many preference methods. FlexPref does not support prioritized composition at the syntax or query processing level.

7.5. Deterministic and Non-Deterministic Answers

In this section, we discuss FlexPref support for preference methods that produce deterministic and non-deterministic answers, summarized in Table V.

7.5.1. Deterministic Answers. FlexPref supports the ability to produce deterministic preference answers, meaning multiple runs of the same preference query over the same data will produce the same answer. At its core, FlexPref is a generic framework that provides a query processing framework for preference methods, while determinism of a preference method is solely a property of the preference method itself. Thus, determinism is controlled by the logic implemented in the extensible FlexPref plugin functions.

7.5.2. Nondeterministic Answers. FlexPref supports the ability to produce nondeterministic preference answers, as long as all nondeterministic of the preference method can be encapsulated in the extensible plugin functions. In other words, a preference method cannot base its nondeterministic behavior on functionality outside of FlexPref, e.g., tuple orders produced by other DBMS operators feeding the FlexPref operators.

8. CASE STUDIES

In this section, we demonstrate the extensibility and usefulness of FlexPref by providing case studies for injecting seven state-of-the-art preference evaluation methods introduced in Section 3.2.2. We chose these case studies carefully to cover a wide spectrum of preference methods. In particular, *skyline* represents transitive dominance-based preference methods, *k-dominance* and *ϵ-dominance* represent non-transitive dominance-based preference methods, *top-k* represents ranking-based preference methods, *top-k dominating* and *k-representative skyline* represent 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).

For each preference method, we first describe its functionality. We then cover the implementation of the general functions described in Section 4, and the query optimization functions described in Section 6. Finally, we give illustrative examples, using the data in Figure 5, for single table, multi-table, and sorted list access. Unless otherwise mentioned, the examples assume the MIN preference over numeric attributes. The details in this section are summarized in the online appendix.

8.1. Case Study 1: Skylines

Given a dataset D, the objective of skyline preference evaluation [Börzsönyi et al. 2001] 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 5(a) object a dominates object e as it is better (i.e., less) in all three dimensions.

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)
Epsilon Dominance
(c,1)
K-Representative Skyline
(a,3), (c,1)

(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) 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. 5. Case study example data

8.1.1. General Function Implementation. A skyline implementation in FlexPref is given below.

Macros

(1) `Default Score`: 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.
(2) `IsTransitive`: Returns *true*; the skyline method exhibits the transitive property.

Evaluation functions

(1) `PairwiseCompare`: Change the score of P to zero *only* if it is dominated by Q, and return the appropriate value (i.e., 1, 0, or -1) 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 4.
(2) `IsPreferredObject`: This function does not need the reference set S to determine if P is a preferred object. Instead, we return *true* if the score of P is one, i.e., P was not dominated by any object.
(3) `AddPreferredToSet`: Append P to the end of set S, and remove any non-skyline objects in S.
(4) `StopSortedEval`: The skyline stopping condition can be based on previous research in distributed skyline query processing [Balke et al. 2004]. 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.

Optimization functions

(1) `EstimateCardinality`: Much previous work has addressed cardinality estimation for the skyline preference method [Chaudhuri et al. 2006; Godfrey 2004; Zhang et al. 2009]. Any of these methods can be used within this function.
(2) `SelectionCommute`: Skyline commutability with selection is conditional upon the selection predicate. Let $C_1(t)$ represent a selection condition over a tuple t; also, let $C_2(p,q)$ represent the given preference criteria that specify whether tuple p is preferred over tuple q (e.g., MIN price and MIN distance). It has been shown that the following condition must hold in order for selection to commute with skyline, which can be verified in quadratic time [Chomicki 2003].

$$\forall p, q[(C_1(q) \wedge C_2(p, q)) \Rightarrow C_1(p)]$$

That is, the selection condition over a "non-preferred" tuple q, logically combined with an AND condition with the preference conditions must imply the selection condition over the preferred tuple p.

(3) `JoinDistribute`: Skyline can distribute over a Cartesian product (i.e., when $\mathcal{J}=\phi$) [Chomicki 2003]. Furthermore, skyline will distribute over an equijoin when the preference objectives in $\mathcal{P}$ apply to attributes in only one of the join input relations (e.g, $\mathcal{F}_{\Theta\mathcal{P}}(R \bowtie_{\mathcal{J}} S) = \mathcal{F}_{\Theta\mathcal{P}_R}(R) \bowtie_{\mathcal{J}} S$). However, skyline will not distribute over an equijoin in the general case when $\mathcal{P}$ applies to attributes in both input relations [Jin et al. 2007; Jin et al. 2010].

8.1.2. Preference Evaluation Example. **Single Table Execution.** Consider the three-dimensional data in Figure 5(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, d, and e 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 5(a).
Multi-Table Execution. In Figure 5(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. 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 5(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.

8.2. Case Study 2: Top-K Dominating

Given a data set D, the objective of top-k dominating preference evaluation [Yiu and Mamoulis 2007] 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 5(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.

8.2.1. General Function Implementation. A top-k dominating implementation in FlexPref is given below.
Macros

(1) `Default Score`: Each object is given a default score of zero.
(2) `IsTransitive`: Returns *true*.

Evaluation functions

(1) `PairwiseCompare`: When it is found 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 this function always returns zero.
(2) `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.
(3) `AddPreferredToSet`: Adds P in sorted order in S, removing the old k^{th} object if applicable.
(4) `StopSortedEval`: One stopping condition 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.

Optimization functions

(1) `EstimateCardinality`: The cardinality is k, since top-k domination can rank objects to produce a total order.
(2) `SelectionCommute`: As discussed in Section 6.3.1, selection does not commute with top-k domination in any case.
(3) `JoinDistribute`: As discussed in Section 6.3.2, given the semantics preference methods designed to return k results, top-k domination does not distribute over a Cartesian product, nor equijoins (which is semantically selection over a Cartesian product).

8.2.2. Preference Evaluation. The top-k domination answer is given in Figure 5(a) assuming $k = 2$. In Figure 5(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 5(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).

8.3. Case Study 3: K-Dominance

Given a data set D and a value k, k-dominance preference evaluation [Chan et al. 2006a] 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 5(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.

8.3.1. General Function Implementation. A k-dominance implementation in FlexPref is given below.

Macros

(1) `Default Score`: As k-dominance does not rank objects, each object can either have a score of one if it is not k-dominated, zero otherwise
(2) `IsTransitive`: Returns *false*, as 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.

Evaluation functions

(1) `PairwiseCompare`: 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.
(2) `IsPreferredObject`: Returns *true* if P's score is 1 (i.e., P is not k-dominated), 0 otherwise.
(3) `AddPreferredToSet`: Appends P to the end of set S.
(4) `StopSortedEval`: A stopping condition 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 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.

Optimization functions

(1) `EstimateCardinality`: There has been no work exploring cardinality estimation of the k-dominance preference method, and providing an in-depth cardinality estimate is outside the scope of this paper. However, for *any* value of k, we can find an *upper bound* estimate by employing skyline cardinality techniques (e.g., see [Chaudhuri et al. 2006; Zhang et al. 2009]).
(2) `SelectionCommute`: Given a d-dimensional dataset, when k=d, k-dominance will commute under the same condition as the skyline method, as they are equivalent in this case. When $k < d$, selection does not commute with k-dominance, as selection performed before preference evaluation may filter objects that can k-dominate objects that qualify for selection. As an example, consider the case of 2-dominance for a datset D with schema (id,d_1,d_2,d_3) and three objects (a,2,3,6), (b,3,2,6), and (c,1,4,2). Given the selection condition $d_2 < 4$, which is "legal" in the case of a skyline, performing $\mathcal{F}_{\Theta\mathcal{P}}(\sigma_{d_2<4}(D))$ will produce $\{a,b\}$ as an answer, while $\sigma_{d_2<4}(\mathcal{F}_{\Theta\mathcal{P}}(D))$ produces $\{\phi\}$ as an answer. In this case pushing selection filtered the key object c that 2-dominates both a and b.
(3) `JoinDistribute`: Given a two join relations R and S with A_R number or attributes in R and A_S number of attributes in S, k-dominance has the same join distribution properties as a skyline when k=A_R+A_S. However, k-dominance does not distribute over a Cartesian product (and hence join) when k <($A_R + A_S$). We consider three cases: (1) $A_R \leq k$ and $A_S \leq k$. Pushing preference evaluation before the join will produce skyline objects as input to the Cartesian product. Per skyline distribution rules, the resulting Cartesian product is also a skyline. However, we can construct an example where the result is not k-dominant. Consider the case where $k = 5$ and R={(1,1,1,1)} and S={(1,2,1,2),(2,1,2,1)} (all three objects are skylines). However, neither tuple in the Cartesian product (1,1,1,1,1,2,1,2) and (1,1,1,1,2,1,2,1) is 5-dominant. (2) $A_R > k$ and $A_S > k$. We can again construct a case where the output is not k-dominant. Consider the previous example from the first case for $k = 3$. Both inputs R and S are 3-dominant, but neither object in the Cartesian product is 3-dominant. (3) $A_R > k$ and $A_S \leq k$ (or vice versa). The exact same argument for case 2 applies to this case.

8.3.2. Preference Evaluation Example. The k-dominance answer is given in Figure 5(a) assuming $k = 2$. In Figure 5(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 5(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).

8.4. Case Study 4: K-Frequency

Given a data set D, k-frequency preference evaluation [Chan et al. 2006b] 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 5(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., $\{d1\}$, $\{d2\}$, $\{d3\}$, $\{d1, d2\}$, $\{d1, d3\}$, $\{d2, d3\}$, $\{d1, d2, d3\}$).

8.4.1. General Function Implementation. A k-frequency implementation in FlexPref is given below.

Macros

(1) `Default Score`: Each object is given a default score of zero.
(2) `IsTransitive`: Returns *true*.

Evaluation functions

(1) `PairwiseCompare`: Dominant sub-dimension counting must be performed carefully for k-frequency. For instance, in Figure 5(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, this function 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. P_{score} is updated 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 this function always returns zero.
(2) `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.
(3) `AddPreferredToSet`: Adds P in sorted order in S, possibly removing the old k^{th} object.
(4) `StopSortedEval`: Uses 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.

Optimization functions

(1) `EstimateCardinality`: The cardinality is k, since k-frequency can produce exactly k answers based on ranking objects in total order.

(2) `SelectionCommute`: k-frequency does not commute with selection, as it ranks objects based on counting dominant subspaces against all other objects in the data set. The basic argument is the same as that of top-k domination.
(3) `JoinCommute`: As discussed in Section 6.3.2, given the semantics preference methods designed to return k results, k-frequency does not distribute over a Cartesian product, nor equijoins.

8.4.2. Preference Evaluation. The k-frequency answer is given in Figure 5(a) assuming $k = 2$. In Figure 5(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 5(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).

8.5. Case Study 5: Top-K

Given a set of data D, top-k preference evaluation [Chaudhuri and Gravano 1999] 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 an object P and returns a single real number as its score. For example, for object a in Figure 5(a) and a monotone function f=($\frac{1}{10}*(d1+d2)+\frac{4}{5}*d3$), a's score is 3.3.

8.5.1. General Function Implementation. A top-k implementation in FlexPref is given below.

Macros

(1) `Default Score`: Each object has a default score of zero.
(2) `IsTransitive`: Returns *true*.

Evaluation functions

(1) `PairwiseCompare`: Top-k does *not* rely on pairwise comparison since an object's score is determined using only its own attributes. Thus, this function returns -2 by default.
(2) `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.
(3) `AddPreferredToSet`: Adds P in sorted order in S, removing the old k^{th} object if applicable.
(4) `StopSortedEval`: A possible stopping condition is based on previous research that defines threshold score for efficient top-k joins over sorted lists [Ilyas et al. 2003]. 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.

Optimization functions

(1) `EstimateCardinality`: The cardinality is k, since top-k can produce exactly k answers based on ranking objects in total order.
(2) `SelectionCommute`: From the extensive previous work in top-k processing, it is semantically correct for top-k to commute with selection [Li et al. 2005].

(3) `JoinCommute`: As discussed in Section 6.3.2, given the semantics preference methods designed to return k results, top-k does not distribute over a Cartesian product, nor equijoins.

8.5.2. Preference Evaluation. The top-k answer is given in Figure 5(a) for $k = 2$, where we aim to minimize scores based on a ranking function that sums all attribute values. In Figure 5(b), pruning 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 5(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.

8.6. Case Study 6: Epsilon Dominance

The epsilon-dominance (abbr. ϵ-dominance) preference method [Xia et al. 2008] alters the concept of traditional skyline dominance to be more flexible. The idea is to increase or decrease the dominance region of each object in the dataset by a constant ϵ[1]. Formally, given two d dimensional points p and q, with a set of weights on each dimension W=$\{w_i|i \in [1,d]$,0 $< w_i \leq 1\}$, p ϵ-dominates q if $\forall i \in [1,d]$, $p[i] \cdot w_i \leq q[i] \cdot w_i + \epsilon$ and $\exists j \in [1,d]$, $p[j] < q[j]$, assuming less is better. The ϵ-dominant answer is given in Figure 5, assuming w_1=0.5, w_2=w_3=1, and ϵ=1.5. In this case, c is the only preference answer, which differs from a traditional skyline answer of $\{a,c\}$, since c ϵ-dominates a.

8.6.1. General Function Implementation. A ϵ-dominance implementation in FlexPref is given below.

Macros

(1) `Default Score`: Each object has a default score of one.
(2) `IsTransitive`: If $\epsilon > 0$, return *false*. In this case, ϵ-dominance loses the transitive property. Otherwise, if $\epsilon \leq 0$, return *true*.

Evaluation functions

(1) `PairwiseCompare`: If P ϵ-dominates Q, return 1. If Q ϵ-dominates P, set the score of P to zero, and return -1. Else, return 0.
(2) `IsPreferredObject`: Return *true* if the score of P is one. Otherwise return *false*.
(3) `AddPreferredToSet`: Append P to the end of set S.
(4) `StopSortedEval`: If $\epsilon \leq 0$, we can use the standard skyline stop condition *stop once there is a complete object **Q** in set **P***. If $\epsilon > 0$, a stopping condition is *stop once there is a complete point **Q** in **P** that is not e-dominated by virtual point **O***. With this condition, we can be certain that any incomplete points will not be dominated by any unseen objects, since the complete point Q is *at least as good* as any incomplete object (i.e., it is better than O).

Optimization functions

(1) `EstimateCardinality`: For the average case, ϵ-dominant cardinality can be estimated by employing skyline cardinality techniques (e.g., see [Chaudhuri et al. 2006; Zhang et al. 2009]). However, refining this estimation for varying values of ϵ remains as future work.
(2) `SelectionCommute`: ϵ-dominance shares similar semantics to skyline when $\epsilon \leq 0$. In this case it shares the same selection commutability rules with skyline. However,

[1] ϵ-dominance is a specific instance of skylines using the more general "substitutable value" (or S-V) preference semantics proposed in [Kießling 2005], where ϵ is similar to the d-value of S-V Semantics

when $\epsilon > 0$, selection does not commute with ϵ-dominance. As ϵ-dominance is non-transitive in this case, the proof is the same as that made for k-dominance, i.e., we can construct a case where a filtered tuple dominates a non-filtered tuple during preference evaluation.

(3) `JoinDistribute`: For any value of ϵ, the ϵ-dominance method shares the same join distribution rules as skyline. This rule holds as the dominance definition for both methods relies on independent pairwise comparability in *every* dimension between two objects. For the case of Cartesian product, performing preference evaluation before the join will guarantee that joined objects are part of the preference answer, as each object's dimensions are compared independent of one another. For the case of equijoin, the same non-distribution for skyline applies to ϵ-dominance.

8.6.2. Preference Evaluation Example. For these ϵ-dominance examples, we assuming w_1=0.5, w_2=w_3=1, and ϵ=1.5. For single-table evaluation, the ϵ-dominant answer is given in Figure 5. In Figure 5(b), ϵ-dominance pruning removes from S tuples (a,3,4), (a,4,3), (a,5,5), and (b,8,8). These pruned tuples are all ϵ-dominated within their join-key groups. Sorted round-robin processing can stop after *five* reads for the data in Figure 5(c). At this point set P contains objects (a,5,3,3) and (b,7,2_), while object O equals (7,3,3) and F equals (5,2,3). The complete object (a,5,3,3) cannot be ϵ-dominated by O.

8.7. Case Study 7: k-Representative Skyline

The k-representative skyline [Lin et al. 2007] is based on the same dominance property of the traditional skyline method, except each object in the preference answer is ranked by the number of objects it dominates. Thus, a total ordering of skyline objects is achieved. As an example, the k-representative skyline preference answer is given in Figure 5, where object a is the top-ranked object with a score of 3, since it dominates objects b, d, and e. Meanwhile, object c is ranked after a with a score of 1, since it only dominates object e.

8.7.1. General Function Implementation. A k-representative skyline implementation in FlexPref is given below.

Macros

(1) `Default Score`: The default score of an object is zero.
(2) `IsTransitive`: Returns *true*.

Evaluation functions

(1) `PairwiseCompare`: If P dominates Q, increase the score of P by one. This function always returns 0, as an object's rank requires pairwise comparison to every other object in the data set.
(2) `IsPreferredObject`: Compare P to object in S, if P is a skyline return *true*. Otherwise, return *false*.
(3) `AddPreferredToSet`: Add P to S in sorted order by the score of P; also remove any non-skyline objects from S.
(4) `StopSortedEval`: The stopping case for k-representative skyline is the same as the standard skyline method.

Optimization functions

(1) `EstimateCardinality`: If the size of the skyline is greater than or equal to k, the k-representative skyline can produce exactly k answers based on the total order ranking of the skyline. Otherwise, cardinality can be estimated using existing estimation techniques for skyline.

(2) `SelectionCommute`: As the k-representative skyline has the same semantics as the skyline, the selection commutability rules are the same as skyline.
(3) `JoinDistribution`: The same join distribution rules for skyline apply to the k-representative skyline.

8.7.2. Preference Evaluation Example. For these examples, we assume k=2, i.e., we want the top-2 representative skyline objects. For single-table evaluation, the k-representative skyline answer is given in Figure 5. In Figure 5(b),pruning removes from S tuples (a,5,5), and (b,8,8), as these tuples can never contribute to the top-2 representative skyline when joined with any counterpart tuple. Sorted round-robin processing can stop after *five* reads for the data in Figure 5(c). At this point set P contains objects (a,5,3,3) and (b,7,2_), while object O equals (7,3,3) and F equals (5,2,3).

9. 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 8. These implementations are denoted Flex$_{SKY}$, Flex$_{KDOM}$, and Flex$_{TK}$, respectively[2]. (2) The *custom* implementations of the skyline block nested loop operator (Cust$_{SKY}$) [Börzsönyi et al. 2001], the sort-first-skyline (Cust$_{SKY-SFS}$) with an added elimination filter step), and the two-scan K-dominance algorithm (Cust$_{KDOM}$) [Chan et al. 2006a]. We also implemented the custom multi-relational skyline join operator (JCust$_{SKY}$) [Jin et al. 2007] 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 [Ilyas et al. 2008]. An exception to this claim is the case involving sorted list access, which we discuss in Section 9.2. Our experiments evaluate four main aspects of FlexPref: (1) multi-table access (Section 9.1), (2) sorted list access (Section 9.2), (3) single table access (Section 9.3), and (4) query optimization (Section 9.4).

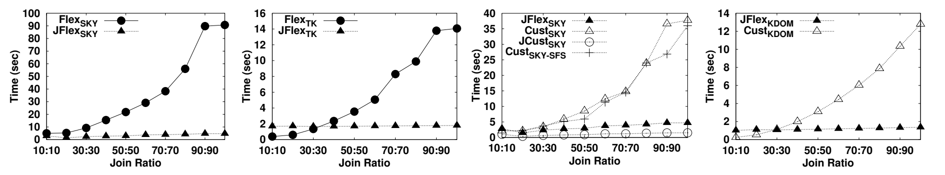
All approaches are implemented in the query processor of the PostgreSQL 8.3.5 open-source database [PostgreSQL]. The experiment machine is an Intel Core2 8400 at 3Ghz with 4GB of RAM running Ubuntu Linux 8.04. We use the generator specified in [Börzsönyi et al. 2001] to generate synthetic data sets for all experiments. Unless otherwise mentioned, the data contain six integer 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. As mentioned in Section 3.2.1, the k in k-dominance is different than that used for top-k. The k in top-k refers to the number of desired answers, while the k in k-dominance represents the *number of dimensions* to use when evaluating dominance. Our performance metric is the *elapsed time* reported by the PostgreSQL EXPLAIN ANALYZE command.

9.1. Multi-Table Join Query

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$_{SKY}$, Cust$_{SKY-SFS}$, and Cust$_{KDOM}$, as well as the custom skyline join algorithm JCust$_{SKY}$. The general SQL signature for this query is:

```
Select * From T1, T2 WHERE T1.id=T2.id
```

[2]We implemented all methods from Section 8, but omit results as the general trend is similar



(a) Sky (b) Top-K (c) Sky (vs Custom) (d) K-Dom (vs Custom)

Fig. 6. Multi-table FlexPref join

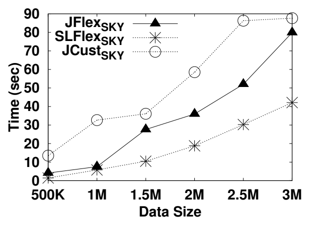
```
Preferring T1.d1 AND T1.d2 AND T1.d3
  AND T2.d1 AND T2.d2 AND T2.d3
```

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.

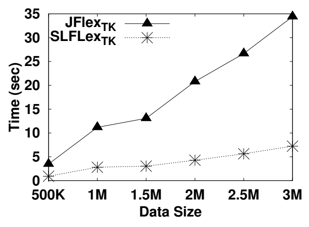
9.1.1. Effect of Pruning. This experiment studies the effect of pruning join inputs in FlexPref's multi-table execution. We study the skyline, and top-k implementations in FlexPref using the naive join approach (abbr. Flex$_{SKY}$, and Flex$_{TK}$) against the optimized pruned approach (abbr. JFlex$_{SKY}$ and JFlex$_{TK}$). For space purposes, we do not discuss the k-dominance implementation, however, it exhibits similar behavior to the skyline case. Figures 6(a) and 6(b) provide the runtimes for the skyline and top-k methods, respectively. 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.

9.1.2. 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$_{SKY}$, Cust$_{SKY-SFS}$, 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 [Jin et al. 2007], JCust$_{SKY}$). Figures 6(c) and 6(d) 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$_{SKY}$, Cust$_{SKY-SFS}$, and Cust$_{KDOM}$ methods that represent an *on-top* approach for the multi-table case. Cust$_{SKY}$, Cust$_{SKY-SFS}$, 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 advantageous 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* pref-



(a) Skyline



(b) Top-K

Fig. 7. Sorted list access

erence 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.

9.2. Sorted List Access

This experiment studies the efficiency of the general sorted access preference evaluation algorithm outlined in Section 5.3 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 signature for this 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
```

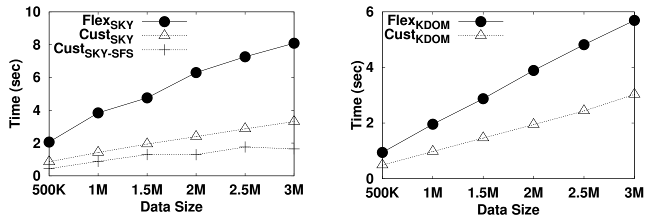
We again omit the `using` clause as multiple preference methods are tested. The join is 1:1 that combines six 2-ary tables T1-T6, each with a primary key id and attribute d; all tables are sorted on d. We compare the FlexPref optimized join implementation (JFlex$_{SKY}$ and JFlex$_{TK}$) to the FlexPref 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 implementation actually reduces to an m-way version of the custom join specified in [Ilyas et al. 2003]. Figure 7 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.

9.3. Single-Table Access

This experiment studies the performance of the skyline and k-dominance preference implementations for a single table access query. The query signature is:

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

Figures 8(a) and 8(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$_{SKY}$, Cust$_{SKY-SFS}$, and Cust$_{KDOM}$). Implemented as user-defined functions, both Cust$_{SKY}$, Cust$_{SKY-SFS}$, and Cust$_{KDOM}$ resemble a *specialized* approach for this experiment as they are designed to read data from a single, unsorted table. As ex-



(a) Skyline (b) K-Dominance

Fig. 8. Single-table preference evaluation

pected, Cust$_{SKY-SFS}$ shows superior performance the generic BNL implementation of Cust$_{SKY}$ as reported in previous work [Chomicki et al. 2003; Godfrey et al. 2005]. We emphasize that 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. Regardless, 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 [Börzsönyi et al. 2001; Chan et al. 2006a].

9.4. FlexPref Query Optimization

This experiment studies the efficiency gains when FlexPref operators can be optimized alongside existing relational operators. Specifically, we investigate performance gains when the selection (Section 9.4.1) and projection (Section 9.4.2) operator can be pushed completely below the single-table FlexPref operator.

9.4.1. Pushing Selection. In this experiment, we study the performance gain when the selection operator can be pushed below the single-table FlexPref operator. The general SQL signature is:

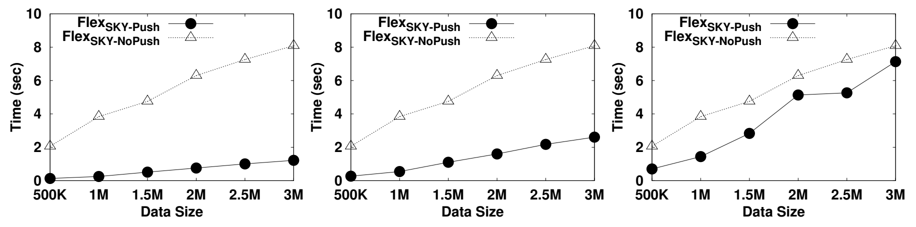
```
Select * From T
Preferring T.d1 AND ... AND T.d6
Using Skyline
Where T.d1 < X
```

The variable X allows for various selectivity ratios. These experiments use the skyline implementation due to its ability to commute with selection (Section 8.1).

Figure 9(a) plots the runtimes for the skyline method as the table cardinality increases from 500K to 3M tuples when the selectivity is set to 10%. When selection is pushed below FlexPref (labeled Flex$_{SKY-Push}$), we see approximately a factor of seven speedup for all data sizes when compared to the case when selection is performed after preference evaluation (labeled Flex$_{SKY-NoPush}$). This performance gain is due selection filtering a large number of tuples before reaching the more expensive FlexPref operator. Figure 9(b) plots the same query for a selectivity of 30%. Even with this higher selectivity ratio, we see approximately a speedup of four when selection is pushed below the FlexPref operator.

9.4.2. Pushing Projection. This experiment studies the performance improvement when the projection operator can be pushed below the single-table FlexPref operator. The general SQL signature is:

```
Select T.d1, T.d2, T.d3 From T
```



(a) 10% Selectivity (b) 30% Selectivity (c) Projection (50% of attributes)

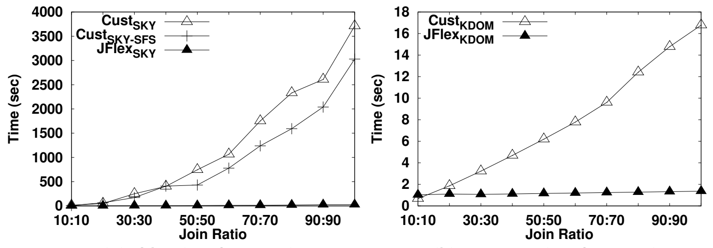
Fig. 9. Query optimization experiments

```
Preferring T.d1 AND ... AND T.d3
Using Skyline
```

For this query, the projection operator removes half of the attributes from the default six-attribute tuples in table T (i.e., T.d4, T.d5, T.d6). This query is legal according to the rules for commuting projection with the FlexPref operator as discussed in Section 6.3.3, as none of the attributes removed by projection take part in preference evaluation. Figure 9(c) provides the query performance numbers when projection is pushed below FlexPref operator implementing the skyline method (labeled Flex$_{SKY-Push}$) and when projection is done after the FlexPref operator (labeled Flex$_{SKY-NoPush}$). Here, we see a small constant speedup when projection is pushed below the FlexPref operator. For both query plans, FlexPref must process all tuples in table T. However, for the Flex$_{SKY-Push}$ approach, the tuples processed by FlexPref are 50% smaller, which is the reason for the better performance.

10. CONCLUSION

This paper presented FlexPref, a general framework for extensible preference evaluation. FlexPref is implemented in the query processor of a database, and supports various 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 (preference selection), joins, and sorted list access. We provided a query optimization framework for FlexPref, as well as a theoretical framework that defines the properties a preference method must exhibit to be implemented in FlexPref. We detailed the implementation of *seven* state-of-the-art preference methods within FlexPref. We also provided experimental evidence that verified the ability of FlexPref to provide efficient query support for arbitrary preference queries. FlexPref lays the groundwork for further generic and extensible support for preference evaluation in databases, including, but not limited to: uncertainty handling, indexing, and integration with aggregate operators.



(a) Sky (vs Custom) (b) K-Dom (vs Custom)

Fig. 10. Three-table join

APPENDIX

This appendix supplements the main body of the paper as follows. Appendix B provides further experimental evaluation of FlexPref. Appendix C provides a single-page implementation summary for all of the case studies presented in Section 8.

B. FURTHER EXPERIMENTAL EVALUATION

This section provides further experimental evaluation of FlexPref. We begin by providing further evidence of FlexPref's usefulness in multi-table join queries. We then evaluate FlexPref using data generated using both anti-correlated and correlated data sets.

B.1. Three-Table Join Query

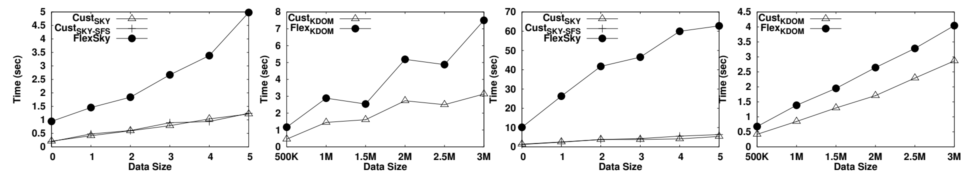
This experiment expands on the multi-table join query presented in Section 9 by using a three-table join query to evaluate FlexPref compared to Cust$_{SKY}$, Cust$_{SKY-SFS}$, and Cust$_{KDOM}$. The SQL signature for this experiment is:

```
Select * From T1, T2, T3 WHERE T1.id=T2.id=T3.id
Preferring T1.d1 AND T1.d2 AND T1.d3
  AND T2.d1 AND T2.d2 AND T2.d3
  AND T3.d1 AND T3.d2 AND T3.d3
```

Figure 10(a) provides the results for the skyline query. In this case the prune optimization of JFlex$_{SKY}$ leads to two orders of magnitude performance speedup over both Cust$_{SKY}$ and Cust$_{SKY-SFS}$. Since this is a non-reductive join, the input to both Cust$_{SKY}$ and Cust$_{SKY-SFS}$ explodes as the number of joins increases. The runtime of both increases by an order of magnitude compared to the two-table join experiment in Section 9. Since both Cust$_{SKY}$ nor Cust$_{SKY-SFS}$ perform preference evaluation after the join, they can do nothing to limit the input. Meanwhile, JFlex$_{SKY}$ scales well with the addition of another join, exhibiting a runtime of 26 seconds in the worst case, compared to 3710 and 3029 seconds for Cust$_{SKY}$ and Cust$_{SKY-SFS}$, respectively. Figure 10(b) plots the results for the k-dominance query. As expected, the overall runtime for Cust$_{KDOM}$ is higher than the two-table query. The runtime for JFlex$_{KDOM}$ is comparatively higher as well, but still exhibits an order of magnitude speedup compared to Cust$_{KDOM}$ for higher join ratios.

B.2. Correlated and Anti-Correlated Data

This experiment explores how FlexPref performs when run on data with correlated and anti-correlated dimensions. For each data set, we re-run the single-table experiment from Section 9. Figures 11(a) and 11(b) report the results for the correlated data. On a whole, the algorithms exhibit better (or same) performance than when using independent data set in Section 9. The custom implementations still show a general trend of besting FlexPref in this setting. As discussed previously, it is not the objective of FlexPref to win against specialized implementations for single-table queries; the previous



(a) Sky (Corr) (b) K-Dom (Cor) (c) Sky (Anti) (d) K-Dom (Anti)

Fig. 11. Single table preference evaluation with correlated and anti-correlated data

join query experiments show that FlexPref is clearly advantageous for more complex preference queries. Figures 11(c) and 11(d) plot the results for the anti-correlated data. These results exhibit the same general trend as the previous experiments.

C. IMPLEMENTATION SUMMARY FOR CASE STUDIES

This section provides a compact implementation summary for all case studies presented in Section 8. Table VI provides this summary; the rows correspond to a preference method, while the columns correspond to each pluggable function of the framework.

REFERENCES

Rakesh Agrawal, Ralf Rantzau, and Evimaria Terzi. 2006. Context-Sensitive Ranking. In *SIGMOD*. 383–394.

Rakesh Agrawal and Edward L. Wimmers. 2000. A Framework for Expressing and Combining Preferences. In *SIGMOD*. 297–306.

Anastasios Arvanitis and Georgia Koutrika. 2012. Towards Preference-aware Relational Databases. In *ICDE*. 426–437.

Wolf-Tilo Balke and Ulrich Güntzer. 2004. Multi-objective Query Processing for Database Systems. In *VLDB*. 936–947.

Wolf-Tilo Balke, Ulrich Güntzer, and Jason Xin Zheng. 2004. Efficient Distributed Skylining for Web Information Systems. In *EDBT*. 597–608.

Don S. Batory. 1986. Extensible Cost Models and Query Optimization in GENESIS. *IEEE Data Engineering Bulletin* 9, 4 (1986), 30–36.

Don S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. 1988. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering* 14, 11 (1988), 1711–1730.

Don S. Batory and Michael V. Mannino. 1986. Panel on Extensible Database Systems. In *SIGMOD*. 187–190.

Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *ICDE*. 421–430.

Michael J. Carey and David J. DeWitt. 1987. An Overview of the EXODUS Project. *IEEE Data Engineering Bulletin* 10, 2 (1987), 47–54.

Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugene J. Shekita, and M. Muralikrishna. 1991. *On Object-Oriented Database Systems*. Springer, Chapter The Architecture of the EXODUS Extensible DBMS, 231–256.

Michael J. Carey and Laura M. Haas. 1990. Extensible Database Management Systems. *SIGMOD Record* 19, 4 (1990), 54–60.

Michael J. Carey and Donald Kossmann. 1997. On saying ”Enough Already!” in SQL. In *SIGMOD*. 219–230.

Chee Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan. 2005. Efficient Processing of Skyline Queries with Partially-Ordered Domains. In *ICDE*. 190–191.

Chee-Yong Chan, H.V. Jagadish, Kian-Lee Tan, Anthony K.H. Tung, and Zhenjie Zhang. 2006a. Finding k-Dominant Skylines in High Dimensional Space. In *SIGMOD*. 503–514.

Chee-Yong Chan, H.V. Jagadish, Kian-Lee Tan, Anthony K.H. Tung, and Zhenjie Zhang. 2006b. On High Dimensional Skylines. In *EDBT*. 478–495.

Kevin Chen-Chuan Chang and Seungwon Hwang. 2002. Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In *SIGMOD*. 346–357.

Surajit Chaudhuri, Nilesh N. Dalvi, and Raghav Kaushik. 2006. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE*. 64.

Surajit Chaudhuri and Luis Gravano. 1999. Evaluating Top-K Selection Queries. In *VLDB*. 397–410.

Jan Chomicki. 2002. Querying with Intrinsic Preferences. In *EDBT*. 34–51.

Jan Chomicki. 2003. Preference Formulas in Relational Queries. *TODS* 28, 4 (2003), 427–466.

Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. 2003. Skyline with Presorting. In *ICDE*. 717–816.

Douglas Comer. 1979. The Ubiquitous B-Tree. *Commun. ACM* 11, 2 (1979), 121–137.

George P. Copeland and Setrag N. Khoshafian. 1985. A Decomposition Storage Model. In *SIGMOD*. 268–279.

Markus Endres and Werner Kießling. 2011. Semi-Skyline Optimization of Constrained Skyline Queries. In *ADC*. 7–16.

Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal Aggregation Algorithms for Middleware. In *PODS*. 102–113.

Parke Godfrey. 2004. Skyline Cardinality for Relational Processing. *Foundations of Information and Knowledge Systems* 2942, 1 (2004), 78–97.

Parke Godfrey, Ryan Shipley, and Jarek Gryz. 2005. Maximal Vector Computation in Large Data Sets. In *SIGMOD*. 229–240.

Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE* 6, 1 (1994), 120–135.

Goetz Graefe and David J. DeWitt. 1987. The EXODUS Optimizer Generator. In *SIGMOD*. 160–172.

Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure For Spatial Searching. In *SIGMOD*. 47–57.

Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. 1989. Extensible Query Processing in Starburst.. In *SIGMOD*. 377–388.

Bernd Hafenrichter and Werner Kießling. 2005. Optimization of Relational Preference Queries. In *ADC*. 175–184.

Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *VLDB*. 562–573.

Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. 2002. Joining Ranked Inputs in Practice. In *VLDB*. 950–961.

Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Supporting Top-k Join Queries in Relational Databases. In *VLDB*. 754–765.

Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *Comput. Surveys* 40, 4 (2008).

Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. 2004. Rank-Aware Query Optimization. In *SIGMOD*. 203–214.

Wen Jin, Martin Ester, Zengjian Hu, and Jiawei Han. 2007. The Multi-Relational Skyline Operator. In *ICDE*. 1276–1280.

Wen Jin, Michael Morse, Jignesh Patel, Martin Ester, and Zengjian Hu. 2010. Evaluating Skylines in the Presence of Equi-joins. In *ICDE*. 249–260.

Navin Kabra and David J. DeWitt. 1999. OPT: An Object-Oriented Implementation for Extensible Database Query Optimization. *VLDB Journal* 8, 1 (1999), 55–78.

Werner Kießling. 2002. Foundations of Preferences in Database Systems. In *VLDB*. 311–322.

Werner Kießling. 2005. Preference Queries with SV-Semantics. In *COMAD*. 16–26.

Werner Kießling, Markus Endres, and Florian Wenzel. 2011. The Preference SQL System - An Overview. *IEEE Data Engineering Bulletin* 34, 2 (2011), 11–18.

Werner Kießling and Gerhard Köstler. 2002. Preference SQL - Design, Implementation, Experiences. In *VLDB*. 990–1001.

Donald Kossmann, Frank Ramsak, and Steffen Rost. 2002. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*. 275–286.

Georgia Koutrika and Yannis Ioannidis. 2004. Personalization of Queries in Database Systems. In *ICDE*. 597–608.

Georgia Koutrika and Yannis Ioannidis. 2005a. Constrained Optimalities in Query Personalization. In *SIGMOD*. 73–84.

Georgia Koutrika and Yannis E. Ioannidis. 2005b. Personalized Queries under a Generalized Preference Model. In *ICDE*. 841–852.

Georgia Koutrika, Evaggelia Pitoura, and Kostas Stefanidis. 2010. Preferences in Databases. In *ICDE*. 1214–1215.

M. Lacroix and Pierre Lavency. 1987. Preferences: Putting More Knowledge into Queries. In *VLDB*. 217–225.

Jongwuk Lee, Gae won You, and Seung won Hwang. 2009. Personlized Top-K Skyline Queries in High-Dimensional Space. *Information Systems* 34, 1 (2009), 45–61.

Justin J. Levandoski, Mohamed Khalefa, and Mohamed F. Mokbel. 2010a. A Demonstration of FlexPref: Extensible Preference Evaluation inside the DBMS Engine. In *SIGMOD*. 1247–1250.

Justin J. Levandoski, Mohamed Khalefa, and Mohamed F. Mokbel. 2010b. FlexPref: A Framework for Extensible Preference Evaluation in Database Systems. In *ICDE*. 828–839.

Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. 2005. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *SIGMOD*. 131–142.

Xuemin Lin, Yidong Yuan, Qing Zhang, and Ying Zhang. 2007. Selecting Stars: The k Most Representative Skyline Operator. In *ICDE*. 86–95.

Volker Linnemann, Klaus Kspert, Peter Dadam, Peter Pistor, R. Erbe, Alfons Kemper, Norbert Sdkamp, Georg Walch, and Mechtild Wallrath. 1988. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions.. In *VLDB*. 294–305.

Guy M. Lohman, George Lapis, Tobin J. Lehman, Rakesh Agrawal, Roberta Cochrane, John McPherson, C. Mohan, Hamid Pirahesh, and Jennifer Widom. 1991. Starburst II: The Extender Strikes Back!. In *SIGMOD*. 447.

Clifford A. Lynch and Michael Stonebraker. 1988. Extended User-Defined Indexing with Application to Textual Databases. In *VLDB*. 306–317.

Steve Olson, Richard Pledereder, Phil Shaw, and David Yach. 1998. The Sybase Architecture for Extensible Data Management. *IEEE Data Engineering Bulletin* 21, 3 (1998), 12–24.

James Ong, Dennis Fogg, and Michael Stonebraker. 1984. Implementation of Data Abstraction in the Relational Database System INGRES. *SIGMOD Record* 14, 1 (1984), 1–14.

Sylvia L. Osborn and T. E. Heaven. 1986. The Design of a Relational Database System with Abstract Data Types for Domains. *TODS* 11, 3 (1986), 357–373.

Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*. 447.

PostgreSQL. http://www.postgresql.org. (????).

Venkatesh Raghavan and Elke Rundensteiner. 2010. Progressive Result Generation for Multi-Criteria Decision Support Queries. In *ICDE*. 733–744.

Berthold Reinwald and Hamid Pirahesh. 1998. SQL Open Heterogeneous Data Access. In *SIGMOD*. 506 – 507.

Berthold Reinwald, Hamid Pirahesh, Ganapathy Krishnamoorthy, George Lapis, Brian T. Tran, and Swati Vora. 1999. Heterogeneous Query Processing through SQL Table Functions. In *ICDE*. 366–373.

Patrick Roocks, Markus Endres, Stefan Mandl, and Werner Kießling. 2012. Composition and Efficient Evaluation of Context-Aware Preference Queries. In *DASFAA*. 81–95.

Leonard D. Shapiro. 1986. Join Processing in Database Systems with Large Main Memories. *TODS* 11, 3 (1986), 239–264.

Jagannathan Srinivasan, Ravi Murthy, Seema Sundara, Nipun Agarwal, and Samuel DeFazio. 2000. Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i. In *ICDE*. 91–100.

Kostas Stefanidis, Georgia Koutrika, and Evaggelia Pitoura. 2011. A Survey on Representation, Composition and Application of Preferences in Database Systems. *TODS* 36, 3 (2011).

Kostas Stefanidis and Evaggelia Pitoura. 2008. Fast Contextual Preference Scoring of Database Tuples. In *EDBT*. 344–355.

Kostas Stefanidis, Evaggelia Pitoura, and Panos Vassiliadis. 2007. Adding Context to Preferences. In *ICDE*. 846–855.

Michael Stonebraker. 1986. Inclusion of New Types in Relational Data Base Systems. In *ICDE*. 262–269.

Michael Stonebraker, Jeff Anton, and Eric N. Hanson. 1987. Extending a Database System with Procedures. *TODS* 12, 3 (1987), 350–376.

Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD*. 340–355.

Yufei Tao, Ling Ding, Xuemin Lin, and Jian Pei. 2009. Distance-based Representative Skyline. In *ICDE*. 892–903.

Akrivi Vlachou, Christos Doulkeridis, and Neoklis Polyzotis. 2011. Skyline Query Processing over Joins. In *SIGMOD*. 73–84.

Florian M. Waas and Joseph M. Hellerstein. 2009. Parallelizing Extensible Query Optimizers. In *SIGMOD*. 871–878.

Florian Wenzel, Markus Endres, Stefan Mandl, and Werner Kießling. 2012. Complex Preference Queries Supporting Spatial Applications for User Groups. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1946–1949.

Annita N. Wilshut and Peter M.G. Apers. 1993. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases* 1, 1 (1993), 103–128.

Raymond ChiWing Wong, Ada WaiChee Fu, Jian Pei, Yip Sing Ho, Tai Wong, and Yubao Liu. 2008. Efficient Skyline Querying with Variable User Preferences on Nominal Attributes. In *VLDB*. 1032–1043.

Tian Xia, Donghui Zhang, and Yufei Tao. 2008. On Skylining with Flexible Dominance Relation. In *ICDE*. 1397–1399.

Man Lung Yiu and Nikos Mamoulis. 2007. Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data. In *VLDB*. 483–494.

Zhenjie Zhang, Yin Yang, Ruichu Cai, Dimitris Papadias, and Anthony Tung. 2009. Kernel-Based Skyline Cardinality Estimation. In *SIGMOD*. 509–522.

Table VI. Implementation summary for case studies

	IsTransitive	DefaultScore	PairwiseCompare	IsPreferredObject	AddPreferredToSet	StopSortedEval
Skyline	*true*	*1*	If P dominates Q, return 1. If Q dominates P, update P_{score} to 0 and return -1. Otherwise, return 0	If object score P_{score} equals 1, return true. Otherwise return false.	Add object P to the end of set S.	Stop once there is a complete object Q in set P
Top-k Domination	*true*	*0*	If P dominates Q, increment P_{score} by 1. Return 0	If the cardinality of S is less than k or P_{score} is superior to the kth object's score in S, return *true*. *False* otherwise.	If S has a cardinality of k, remove the kth object from S. Add P to S in sorted order by P_{score}.	Stop once there are k complete objects in set P.
K-Dominance	*false*	*1*	If P k-dominates Q, return 1. If Q k-dominates P, set $P_{score} = 0$, return -1. Else return 0.	If object score P_{score} equals 1, return true. Otherwise return false.	Add object P to the end of set S.	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.
K-Frequency	*true*	*0*	Increment P_{score} based on the *distinct* subdimensions where Q dominates P. Return 0.	If $\|S\| < k$ or P_{score} is superior to the kth object's score in S, return *true*. Otherwise, return *false*.	If S has a cardinality of k, remove the kth object from S. Add P to S in sorted order by P_{score}.	Stop once there is a complete object Q in set P.
Top-K	*true*	*0*	Return -1	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*.	If S has a cardinality of k, remove the kth object from S. Add P to S in sorted order by P_{score}.	Stop once P contains k complete objects that have scores $\leq$ a given threshold value T. T = MIN(f(O[1],F[2] $\cdots$,F[n]), f(F[1],O[2], ...,F[n]), f(F[1],F[2] $\cdots$,O[n])).
Epsilon Dominance	if $\epsilon > 0$. *true* else *false*	*0*	If P ϵ-dominates Q, return 1. If Q ϵ-dominates P, set the score of P to zero, and return -1. Else, return 0.	Return *true* if the score of P is one. Otherwise return *false*.	Append P to the end of set S.	If $\epsilon \leq 0$, stop once there is a complete object Q in set P. If $\epsilon > 0$, stop once there is a complete point Q in P that is not e-dominated by virtual point O.
k-Representative Skyline	*true*	*0*	If P dominates Q, increment P_{score} and return 0	Compare P to object in S, if P is a skyline return *true*. Otherwise, return *false*.	Add P to S in sorted order by the score of P; also remove any non-skyline objects from S.	Same as the *skyline* method.