PrefDB: Bringing Preferences Closer to the DBMS

Anastasios Arvanitis
National Technical University of Athens, Greece
anarv@dblab.ntua.gr

Georgia Koutrika
IBM Almaden Research Center
San Jose, CA, USA
gkoutri@us.ibm.com

ABSTRACT

In this demonstration we present a preference-aware relational query answering system, termed *PrefDB*. The key novelty of *PrefDB* is the use of an extended relational data model and algebra that allow expressing different flavors of preferential queries. Furthermore, unlike existing approaches that either treat the DBMS as a black box or require modifications of the database core, *PrefDB*'s hybrid implementation enables operator-level query optimizations without being obtrusive to the database engine. We showcase the flexibility and efficiency of *PrefDB* using *PrefDBAdmin*, a graphical tool that we have built aiming at assisting application designers in the task of building, testing and tuning queries with preferences.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Applications

Keywords

Query Personalization, Preferences

1. INTRODUCTION

Database systems support a boolean query model where conditions are considered hard and a non-empty answer is returned only if it satisfies all of them. However, this model is often too strict. For example, consider a movie rental application and a user looking for 2011 movies with Adam Sandler and Tom Hanks. Since no such movies exist, it may be preferable to consider the query criteria as soft (i.e., preferences) and return 2011 movies with any of the two actors. Incorporating preferences into database queries is also important in query personalization. For instance, taking into account a user's preferences for comedies can help focus her query about recent movies, which would otherwise return too many results.

Several approaches to integrating preferences into database queries have been proposed and can be roughly divided into two categories. Plug-in approaches (e.g., [4, 5]) provide a query translation layer on top of the database that allows expressing and executing queries with preferences as ordinary queries over ordinary database relations. However, the way preferences will be translated is typically hard-coded in the application logic, offering limited control to the application developer. Further, each preferential query might require several ordinary queries to be expressed, thus affecting query performance. On the other hand, native methods (e.g., [3, 7]) inject a new preference operator into the database engine. Each approach exhibits better performance compared to a plug-in approach, but has limited flexibility, because the supported query type (e.g., top-k [7] or skyline [3]) is now hard-wired into the preference operator. One solution to enhance flexibility is to implement multiple preference methods in a single database system. To make this practical, FlexPref [6], built inside the query processor, allows different preference evaluation methods to be implemented in a single extensible source code, and subsequently be used inside database queries.

In an effort to deal with the aforementioned limitations, we have followed a different path and we have developed *PrefDB*, a preference-aware relational system that transparently and efficiently evaluates queries with preferences [2]. *PrefDB* uses an extended, preference-aware, relational data model and algebra that allow explicitly formulating different flavors of preferential queries. Database tuples have scores and confidences that reflect preferences. Preferences appear inside queries as first-class citizens with the help of a new operator: the prefer operator has as inputs a relation and a preference on this relation and outputs the relation with new scores and confidences. In contrast to other approaches (e.g., [3, 6, 7]), the prefer operator does not disqualify tuples (i.e., preferences are treated as soft constraints), hence it does not restrict the type of queries with preferences that can be supported. Another difference with native approaches is *PrefDB*'s hybrid implementation, which is closer to the database than plug-in approaches enabling finer-grained optimizations, yet not as obtrusive as native ones.

By pushing preferences into the database, *PrefDB* offers improved expressiveness and simplified engineering for the application designer. In order to demonstrate these capabilities, in the spirit of visual tools that allow application designers to build and evaluate queries over traditional databases, we have developed *PrefDBAdmin*: a graphical tool that provides similar (but extended) functionality for queries with preferences over *PrefDB*. *PrefDBAdmin* allows an application designer to perform the following tasks:

- Manage user preferences, group them into profiles and select which ones and how they will be used in queries.
- Build and execute preferential queries, select among a set of available execution strategies and configure various query parameters such as the expected number and type of results (i.e., top-k, skyline), score and confidence thresholds, and so forth.
- Inspect the query execution through a built-in console, explore the preference-aware query plan followed, and browse statistics and profiling information available for their queries.

In this demo, we will use *PrefDBAdmin* as a means to expose the functionality and flexibility of *PrefDB*. We will also showcase how

p_1 : `Alice rated the movie 'Scoop' with 8/10`
p_2 : `Alice's favorite director is Woody Allen`
p_3 : `She prefers higher-rated films if voted by several users`
p_4 : `Alice loves comedies`

Figure 1: Example set of a user's preferences

application designers can build, test and tune preferential queries.

Outline. Section 2 provides an overview of the preference and query models used in *PrefDB* and Section 3 describes the *PrefDB* system. In Section 4 we present *PrefDBAdmin*, and in Section 5 we describe our demonstration scenario.

2. A PREFERENCE-AWARE DBMS

2.1 Preference Model

First, we will present the preference model that we follow. A preference is defined on a relation R as a triple with (*i*) a *conditional part* that describes the tuples affected by the preference, (*ii*) a *scoring part* that scores these tuples, and (*iii*) a *confidence part* as a degree of certainty for the preference. Note that while prior preference models have studied the first two aspects, our approach is the first that couples them in a concise, powerful, quantitative model. In addition, confidence is an inherent preference property captured for the first time in our model. Confidence represents a measure of the uncertainty that comes with preferences due to different learning paths. For instance, a preference for the director 'Woody Allen' that is explicitly expressed by a user (e.g., using a 'Like' button) may have a higher confidence value than a preference derived by the system based on the user's movie selections (being possibly less certain). Furthermore, preferences can be either *atomic* or *generic*. Atomic preferences involve exactly one tuple and capture essentially user ratings. Generic preferences are usually inferred by the system and they are set-oriented, i.e., they are bound to a set of tuples that satisfy a condition.

To illustrate our model, consider an online video rental service and a user, Alice. Alice has rated the film 'Scoop' with 8/10 (preference p_1 in Figure 1). This atomic preference can be expressed as $p_1[MOVIES] = (\sigma_{m\_id=m_5}, 0.8, 1)$, where the selection condition, $\sigma_{m\_id=m_5}$, specifies the movie of interest, and 0.8 is the score. The confidence is 1, since p_1 represents an explicit rating by Alice. The system has also extracted generic preferences for Alice based on her ratings. Preference p_2 is expressed as $p_2[DIRECTORS] = (\sigma_{director=\text{'}W.Allen\text{'}}, 1, 0.8)$ whereas preference p_3 can be captured as $p_3[MOVIES] = (\sigma_{votes\geq 100}, S_r(rating), 0.8)$. Note how confidence (0.8) in both p_2 and p_3 is lower than in p_1 because p_2 and p_3 have not been directly expressed by Alice. Furthermore, the scoring part of a preference can be a function that assigns different scores to the tuples affected by the preference based on the tuple values. For instance, in p_3, S_r is a scoring function that assigns higher scores to higher-rated movies. Different scoring functions are part of the system for expressing preferences.

PrefDB allows formulating rich preferences (e.g., conditional, multi-relational, etc). We expect this expressivity to be particularly useful, especially for applications with large database schemas that go beyond the typical single-table "find me a restaurant" scenario. For instance, in a movie rental application, preferences may be expressed using conditions that span over several tables (e.g. a preference for 'comedies starring Meg Ryan' would combine conditions on $GENRES$ and $ACTORS$ joined with the $MOVIES$ relation).

2.2 Extended Data Model and Algebra

We extend database tuples in order to reflect preferences as follows. We define a p-relation as a relation with two additional attributes: a score and a confidence. Tuple scores and confidences can be either (*i*) assigned by evaluating user preferences or (*ii*) passed over through a relational operator (e.g., join).

m_id	title	year	duration	d_id	score	conf
m_1	Gran Torino	2008	116	d_1	0.77	1
m_2	Wall Street	1987	126	d_3	0.83	1
m_3	Million Dollar Baby	2004	132	d_1	0.85	1
m_4	Match Point	2005	124	d_2	$\perp$	0
m_5	Scoop	2006	96	d_2	0.78	1

(a) $MOVIES$

d_id	director	score	conf
d_1	C. Eastwood	0.8	1
d_2	W. Allen	0.9	1
d_3	O. Stone	$\perp$	0

(b) $DIRECTORS$

m_id	d_id	score	conf
m_1	d_1	0.785	2
m_2	d_3	0.83	1
m_3	d_1	0.825	2
m_4	d_2	0.9	1
m_5	d_2	0.84	2

(c) $MOVIES \bowtie DIRECTORS$

m_id	score	conf
m_1	0.885	2
m_2	0.83	1
m_3	0.924	2
m_4	0.997	1
m_5	0.889	2

(d) $\lambda_{p_a,F}(MOVIES)$

Figure 2: Evaluating preferences and joins on p-relations

When more than one score-confidence pair for a tuple may exist, we combine them into a final score and confidence with the help of an aggregate function. Several aggregate functions might apply, such as the sum, the average, the min/max, and so forth, reflecting different philosophies on how to combine partial values into a total score/confidence. Selecting the most appropriate function depends on the application scenario. In Examples 1 and 2, we will use an aggregate function F that sums the partial confidence values and calculates a weighted average of the individual scores using the corresponding confidence values as weights.

Further, we extend all relational operators to handle p-relations. For example, the extended join produces the same results as the traditional join but the results also carry scores/confidences that reflect the scores/confidences of the joined tuples based on the aggregate function used. To illustrate, let us consider an example:

Example 1. Figures 2(a) and 2(b) show two p-relations $MOVIES$ and $DIRECTORS$. As depicted, both relations have some already attached scores/confidences reflecting the user's preferences, where $\perp$ denotes that no preference is relevant for the specific tuple and it is typically the default score for a tuple. Figure 2(c) shows the result of evaluating $MOVIES \bowtie DIRECTORS$ using F.$\Box$

Finally, in order to evaluate preferences on tuples, we introduce a special prefer operator $\lambda_{p,F}(R)$. The prefer operator evaluates a preference p on R using the aggregate function F. In particular, the prefer operator is executed on all tuples satisfying the preference condition of p, calculates their scores based on the scoring part of p and, finally, updates tuple scores and confidences using F.

Example 2. Consider the p-relation $MOVIES$ in Figure 2(a) and preference: $p_a[MOVIES] = (\sigma_{year\geq 2000}, S_m(year, 2012), 1)$, where $S_m(year, x) = year/x$. Figure 2(d) shows the resulting p-relation after evaluating $\lambda_{p_a,F}(MOVIES)$. $\Box$

2.3 Preferential Queries

A *preferential query* combines p-relations, extended relational and prefer operators and returns a set of tuples that satisfy the boolean query conditions along with their score and confidence values that have been calculated after evaluating all prefer operators on the corresponding relations. The better a tuple matches preferences and the more (or more confident) preferences it satisfies, the higher final score and confidence values it will have respectively.

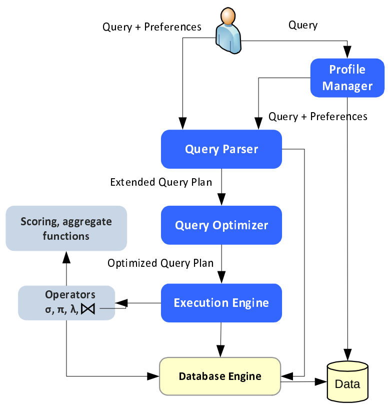
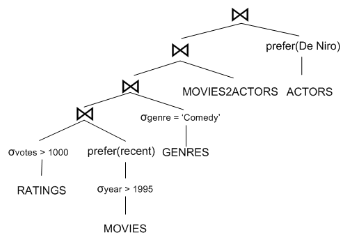
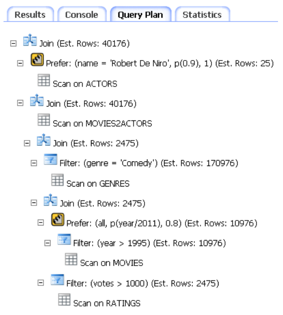


Figure 3: System Architecture



(a) Extended Query Plan



```
⊟ Join (Est. Rows: 40176)
  ⊟ Prefer: (name = 'Robert De Niro', p(0.9), 1) (Est. Rows: 25)
      Scan on ACTORS
  ⊟ Join (Est. Rows: 40176)
      Scan on MOVIES2ACTORS
    ⊟ Join (Est. Rows: 2475)
      ⊟ Filter: (genre = 'Comedy') (Est. Rows: 170976)
          Scan on GENRES
      ⊟ Join (Est. Rows: 2475)
        ⊟ Prefer: (all, p(year/2011), 0.8) (Est. Rows: 10976)
          ⊟ Filter: (year > 1995) (Est. Rows: 10976)
              Scan on MOVIES
        ⊟ Filter: (votes > 1000) (Est. Rows: 2475)
            Scan on RATINGS
```

(b) Query Profiler

Figure 4: Preference-aware Query Plan

PrefDB offers two alternative query options: preferences can be provided along with the input query or the system can enrich a non-preferential query with related preferences. In both cases, *PrefDB* builds a preferential query expressed in our extended relational algebra. In the following, we provide some query examples.

Example 3. Let us consider our online video rental service and that the preferences known for Alice are those depicted in Figure 1. Assume Alice is looking for 2011 movies. In order to return results that are of interest to Alice, the application uses the following query that specifies 2011 movies with the highest scores according to Alice's preferences.

$Q_1 : top(k, score)\pi_{title,director}\{\lambda_{p_3}\sigma_{year=2011}(MOVIES) \bowtie \lambda_{p_4}(GENRES) \bowtie \lambda_{p_2}(DIRECTORS)\}$

where $top(k, score)$ filters the k most highly ranked results. □

Example 4. Let us assume that the application provides social recommendations by enriching the movies satisfying Alice's preferences (p_2, p_3, p_4) with movies suggested by her friend Bob. Say that Bob's preferences are about movies (p_5) and about actors (p_6). The following query selects the most confident results for Alice and the most highly-ranked results for Bob, and aggregates the two sets of movies into a single list of movies for Alice.

$Q_2 : \{\pi_{title}\sigma_{conf>\tau_c}\lambda_{p_3}(MOVIES) \bowtie \lambda_{p_4}(GENRES) \bowtie \lambda_{p_2}(DIRECTORS)\} \bigcup \{\pi_{title}\sigma_{score>\tau_s}\lambda_{p_5}(MOVIES) \bowtie MOVIES2ACTORS \bowtie \lambda_{p_6}(ACTORS)\}$ □

3. PREFDB OVERVIEW

In this section, we briefly describe the basic modules of the *PrefDB* system implementation (depicted in Figure 3). For a more detailed description of our query optimization and processing strategies, we refer the interested reader to [2].

The *profile manager* aggregates and stores user preferences and profiles into the *PrefDB* database. Preferences can be explicitly defined through our graphical tool *PrefDBAdmin*, or they can be recorded from user actions (e.g., clickthrough data). Thereby, when preferences are not provided along with the input query, the profile manager applies the preference selection algorithm proposed in [5] and selects all preferences, which are related with the user and the query context. For instance, if the user is searching for a recent movie, *PrefDB* would fetch all preferences defined on $MOVIES$, as well as joined relations such as $DIRECTORS$ or $GENRES$.

The *query parser* takes as input the user query and related preferences and generates a preferential query expressed in the extended relational algebra. Additionally, the query parser generates a baseline *extended query plan* keeping the order of the operators as defined in the input query. Figure 4(a) illustrates one such query plan.

The *query optimizer* rewrites the input query plan into a more efficient one based on the algebraic properties of the prefer operator. For example, given the processing cost required for calculating and aggregating scores, a primary optimization goal is to reduce the input size given to preference evaluation. Therefore, prefer operators will be placed above select and project operators. Further, using the constructed plan as a basis, the optimizer examines alternative execution plans, and estimates their costs based on a cost model that we have developed that incorporates preference evaluation.

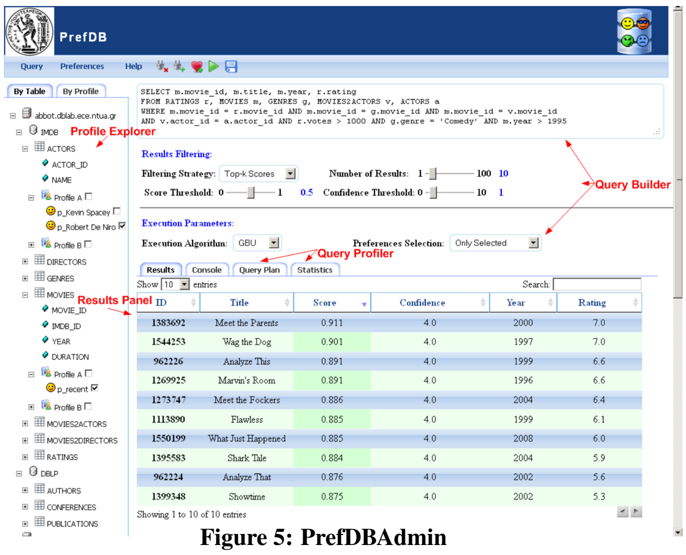
The *execution engine* receives from the query optimizer the selected query plan and executes it. It supports a set of query evaluation strategies that we have developed. One approach leverages the native query engine by delegating the non-preference query part to be directly executed by the database. Subsequently, the remaining prefer operators are evaluated on the qualifying tuples, similarly to plug-in approaches. Another approach follows a more holistic processing strategy, trying to group together operators or saving intermediate materializations whenever possible.

We have built *PrefDB* on top of PostgreSQL, by implementing the prefer and extended relational operators as UDFs in pgSQL. However, our system can be easily ported to other relational database systems as well. Further, we emphasize that in contrast with previous systems [6, 7], our implementation does not require any modification of the database source code, making it a practical solution for real-world applications that seek to integrate user preferences into their business logic, especially when hacking the underlying DBMS is not possible or desirable.

4. INTERACTING WITH PREFDB

In this section, we will walk through *PrefDBAdmin*, a graphical tool that we have implemented for assisting database administrators and application developers (collectively referred to as 'the designer' hereafter) in building, testing and tuning their queries with preferences over *PrefDB*. *PrefDBAdmin* consists of the following modules, as illustrated in Figure 5.

Profile Explorer. The Profile Explorer (left part of Figure 5) is a tree representation of the databases managed through the tool. For each database, the designer can view its tables, and for each database table, its columns and the preferences attached to it. For example, in Figure 5 the table $ACTORS$ has two related preferences. Preferences are organized in profiles. Designers can select to browse preferences either by table or by the profile they belong to. We distinguish between two profile categories: *user profiles* are automatically generated by aggregating the preferences for each user, whereas *test profiles* are manually constructed by the designer to be used for experimentation and tuning of queries with preferences. User profiles can only be used without modifications in queries and they are anonymized to avoid associating preferences with specific users. *PrefDBAdmin* allows designers to create new preferences or



```
SELECT m.movie_id, m.title, m.year, r.rating
FROM RATINGS r, MOVIES m, GENRES g, MOVIES2ACTORS v, ACTORS a
WHERE m.movie_id = r.movie_id AND m.movie_id = g.movie_id AND m.movie_id = v.movie_id
AND v.actor_id = a.actor_id AND r.votes > 1000 AND g.genre = 'Comedy' AND m.year > 1995
```

Results Filtering:

Filtering Strategy: Top-k Scores   Number of Results: 1 – 100 10

Score Threshold: 0 – 1 0.5   Confidence Threshold: 0 – 10 1

Execution Parameters:

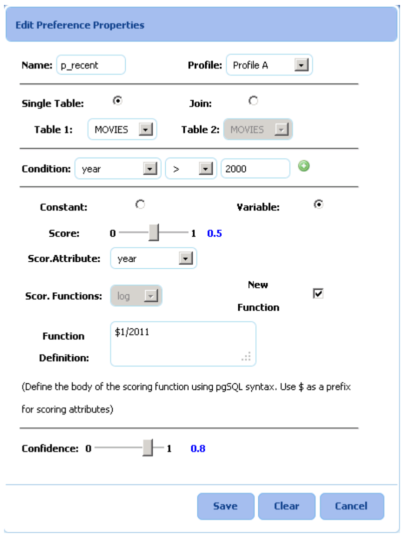
Execution Algorithm: GBU   Preferences Selection: Only Selected

Show 10 entries   Search:

ID	Title	Score	Confidence	Year	Rating
1383692	Meet the Parents	0.911	4.0	2000	7.0
1544253	Wag the Dog	0.901	4.0	1997	7.0
962226	Analyze This	0.891	4.0	1999	6.6
1269925	Marvin's Room	0.891	4.0	1996	6.6
1273747	Meet the Fockers	0.886	4.0	2004	6.4
1113890	Flawless	0.885	4.0	1999	6.1
1550199	What Just Happened	0.885	4.0	2008	6.0
1395583	Shark Tale	0.884	4.0	2004	5.9
962224	Analyze That	0.876	4.0	2002	5.6
1399348	Showtime	0.875	4.0	2002	5.3

Showing 1 to 10 of 10 entries

Figure 5: PrefDBAdmin



Name: p_recent   Profile: Profile A

Single Table: ● Join: ○

Table 1: MOVIES   Table 2: MOVIES

Condition: year > 2000

Constant: ○ Variable: ●

Score: 0 – 1 0.5

Scor.Attribute: year

Scor. Functions: log   New Function ☑

Function Definition: $1/2011

(Define the body of the scoring function using pgSQL syntax. Use $ as a prefix for scoring attributes)

Confidence: 0 – 1 0.8

Figure 6: Editing Preferences

edit existing ones inside test profiles using the Preference Editor.

Preference Editor. The Preference Editor (Figure 6) allows the designer to create and edit preferences. The designer can customize several preference properties, such as the condition, the scoring part and the confidence. The designer can either specify a constant score for all relevant tuples, or select among a set of predefined scoring functions or edit the body of the scoring function to be used.

Query Builder. The designer can build queries with preferences to be executed by *PrefDB*. Starting from the Profile Explorer, the user of *PrefDBAdmin* can select which preferences or profiles will be used in a query. She can manually specify the SQL part and customize query parameters such as the desired number of results to be returned, the minimum score and confidence in the results, the filtering strategy, i.e., top-k or skyline (top part of Figure 5). The designer can further choose the execution algorithm that will be used among the different execution algorithms developed for the *PrefDB* system and a plug-in implementation [2]. Thereby, the designer can compare *PrefDB*'s hybrid execution performance to plug-in alternatives for the same query. Finally, the designer can execute the query or save the query specification (using the options given in the menu bar). Using the same menu, the designer can also load a previously saved query specification.

Results Panel. Upon completion of the query execution, the designer can explore different subsets of the results. For instance, she can sort results in different orders, search over the results and modify the number of returned results and score thresholds.

Query Profiler. The system allows users to explore the extended query plan that will be followed, along with cost estimations regarding the individual operations that will be executed (Figure 4(b)). During query execution, the designer can monitor the query execution using the built-in console. Upon completion of the query, the designer can use the statistics tab and view useful profiling information, such as the CPU time actually spent on individual operations and the sizes of intermediate results, and compare them to the predicted values. Finally, the designer can explore historical statistics for the queries that she has recently executed. For example, she can view for the past queries the time spent for preference evaluation and the time spent on other operations such as joins and filters.

5. DEMONSTRATION SCENARIO

In this demonstration we showcase the functionality of the *PrefDB* framework and the use of *PrefDBAdmin*. In particular, our goal is to allow the demo participants to experiment with the features of *PrefDB* on a movie recommendation application that we have developed, based on on a database instance that we have extracted from IMDB [1]. For the demo purposes, we have compiled a set of template user profiles, each one consisting of several preferences. Using *PrefDBAdmin*, attendees will have the option to use any of the available profiles, or construct new ones to be used in queries.

We will navigate through the database schema and user profiles and show how easily preferences can be captured. We will showcase different preference types such as atomic, conditional or multi-relational ones. Further, the demo attendees will be able to edit preferences, create new ones, and combine them into profiles.

Next, we will demonstrate how queries with preferences can be built using *PrefDBAdmin*. Participants will have the choice to either let the system use the most relevant preferences, or manually select among a set of available preferences and profiles. Further, the attendees will be able to specify a query, configure query parameters and finally execute the query. By changing the query options and re-running the query, participants should be able to experiment with different filtering strategies and examine the impact on the relevancy of the query results through the results panel.

Finally, we will showcase the query profiler, where the participants will be able to inspect the query execution and explore statistics and profiling information. We will focus on selected scenarios that show how the performance is affected by the query complexity and the number of preferences used.

6. REFERENCES

[1] IMDB movie database. http://www.imdb.com.

[2] A. Arvanitis and G. Koutrika. Towards preference-aware relational databases. In *ICDE*, 2012 (to appear).

[3] J. Chomicki. Preference formulas in relational queries. *TODS*, 28(4):427–466, 2003.

[4] W. Kießling and G. Köstler. Preference SQL - design, implementation, experiences. In *VLDB*, pages 990–1001, 2002.

[5] G. Koutrika and Y. E. Ioannidis. Personalization of queries in database systems. In *ICDE*, pages 597–608, 2004.

[6] J. Levandoski, M. Mokbel, and M. Khalefa. FlexPref: A framework for extensible preference evaluation in database systems. In *ICDE*, pages 828–839, 2010.

[7] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.