Towards Preference-aware Relational Databases

Anastasios Arvanitis #1, Georgia Koutrika *2

# National Technical University of Athens, Greece
1anarv@dblab.ntua.gr
2gkoutri@us.ibm.com

Abstract—In implementing preference-aware query processing, a straightforward option is to build a plug-in on top of the database engine. However, treating the DBMS as a black box affects both the expressivity and performance of queries with preferences. In this paper, we argue that preference-aware query processing needs to be pushed closer to the DBMS. We present a preference-aware relational data model that extends database tuples with preferences and an extended algebra that captures the essence of processing queries with preferences. A key novelty of our preference model itself is that it defines a preference in three dimensions: the tuples affected, their preference scores and the credibility of the preference. Our query processing strategies push preference evaluation inside the query plan and leverage its algebraic properties for finer-grained query optimization. We experimentally evaluate the proposed strategies. Finally, we compare our framework to a pure plug-in implementation and show its feasibility and advantages.

I. INTRODUCTION

The concept of preference-aware query processing is dominant in many applications where there is a matter of choice among alternatives, ranging from query personalization [7], [16], [19], recommendations [3] to multi-criteria decision making [6], [9], IR-style queries [25] and so forth. Several implementations of preference-aware query processing exist and they can be broadly divided into two categories.

Plug-in approaches are built on top of the database engine (e.g., [6], [17], [19]). Their main advantage is that they require no modification of the database code hence they leverage existing, invested, technology. On the downside, treating the DBMS as a black box affects performance. Plug-in query execution plans are composed outside the database engine and describe how queries with preferences are translated to conventional queries and executed over the DBMS. With no access to internal database operations like joins and selections, only coarse-grained query optimizations, such as reducing the number of queries sent to the DBMS, are possible. On the other hand, native approaches directly manipulate the database engine by injecting new preference operators (e.g., [7], [22]) and building internal query plans that can be dynamically optimized at the operator level. However, modifying the database engine may not be feasible or practical in real life. Moreover, existing native methods are focused to limited query types, such as top-k [22] or skylines [7], and they are not generally applicable to any kind of query with preferences.

Motivated by the above, in this paper, we aim at combining the pros of both approaches without their cons and we propose a novel approach to preference-aware querying that is:

- in-between: its implementation is closer to the database than plug-in approaches hence using finer-grained optimizations, yet it is not as obtrusive as native approaches.
- flexible: it supports formulating and handling different flavors of preferential queries in a flexible, generic way.

We propose a preference-aware framework where preferences appear inside queries as first-class citizens and preference evaluation is captured as a special operator that can be combined with the classical relational operators. Our framework has three components: a preference model, a preference-aware relational data model and a preference-aware algebra.

We define a preference along three dimensions: (conditional) which tuples are affected, (scoring) how they are scored and (confidence) how confident the scores are. Note that while prior preference models (e.g., [4], [19]) have studied the first two aspects, our representation is the first that concisely couples them in a quantitative way and captures rich preferences. In addition, confidence, an inherent preference property, is new to our model. For instance, confidence allows us to distinguish a preference for the director Clint Eastwood that the user has explicitly expressed (for which we are confident that it holds) from a preference learnt by the system from the user’s movie selections (hence it may be less certain).

The core of our framework comprises a preference-aware relational data model and algebra. Our data model attaches scores and confidences to tuples that reflect preferences that have been directly evaluated on a relation or passed over by other relations through relational operators, such as joins and unions. Our algebra comprises all standard relational operators extended to handle scores and confidences. For example, the join operator will join two tuples and compute a new score-confidence pair by combining the scores and confidences that come with the two tuples. Finally, a new operator is introduced: the prefer operator evaluates a preference on a relation and outputs a relation with updated scores and confidences.

Our framework allows us to naturally blend preferences into queries. We provide several examples of preferential queries that showcase its flexibility and expressivity. An important feature of our work is that we separate the concept of preference evaluation from the concept of tuple filtering. During preference evaluation, both the conditional and the scoring part of a preference are used. The conditional part acts as a ‘soft’ constraint that determines how tuples are scored and it does not disqualify tuples from the query result. Tuple filtering conceptually follows preference evaluation. In this phase, tuples can be filtered depending on the answer expected, e.g., top-k results [22] based on score or confidence, all results ranked, not-dominated ones [7] or those that satisfy a minimum number of preferences [19] or the
most confident preferences using our model, etc. Previous works have tackled preference evaluation and filtering as one operation and developed techniques focusing on particular types of queries based on the filtering strategy followed. However, preference evaluation and tuple filtering are not just conceptually different. Distinguishing them has several benefits in practice.

More specifically, we are able to clearly isolate and study the properties of preference evaluation in the context of relational query processing and develop generic processing and optimization techniques that are independent of how preferred tuples may be filtered. Our processing strategies do not translate preferences into traditional relational constructs as plug-in approaches do. Instead, they leverage the algebraic properties of the prefer operator, which is pushed inside the query plan enabling finer-grained query optimization. At the same time, our methods do not require modifications of the database engine code, which in real life may be impractical, keeping the advantages of a pure plug-in approach. For example, parts of the query that are not explicitly related to preferences are executed leveraging the native query optimizer.

We evaluate our query processing techniques and we compare them with two implementations of the plug-in approach. The basic idea of the plug-in strategy is that it rewrites a query with preferences by incorporating the preferences as standard query conditions, executes the query, scores and aggregates the partial results into a single query answer. Note that our algorithms are not directly comparable to top-k or skyline operators and algorithms. RankSQL [22] extends the relational algebra with a new operator called rank that enables pipelining and hence optimizing top-k queries. In order to facilitate the integration of any preference operator (e.g., top-k, skyline) into the database engine with minimal effort, FlexPref is a system that allows registering prepared preference methods through an API that allows defining rules for determining the most preferred tuples [21].

Regarding preference integration and processing, a straightforward approach is to translate queries with preferences to conventional queries and execute them over the DBMS [10], [16], [19], [20], [23]. Query translation conceptually involves the following steps: (Rewrite) the preferences are integrated as standard query conditions in the query producing a set of new queries, (Materialize) the new queries are executed and (Aggregate) the partial results are combined into a single ranked list. This is also the approach followed by Preference SQL [17] for implementing the special operator (BMO) proposed for embedding preference relations into relational queries [11].

Apart from BMO, other flavors of preferential queries include winnow [7], skyline [6] and top-k queries [9], [13], [22]. Several special evaluation algorithms for answering such queries on-top of the DBMS have been proposed (e.g., for top-k results [9], [13] or the skyline [6], [24]). However, these algorithms as well as query translation methods are plug-in, i.e., implemented outside the DBMS. Few native implementations modify the database engine by adding specific physical operators and algorithms. RankSQL [22] extends the relational algebra with a new operator called rank that enables pipelining and hence optimizing top-k queries. In order to facilitate the integration of any preference operator (e.g., top-k, skyline) into the database engine with minimal effort, FlexPref is a system that allows registering prepared preference methods through an API that allows defining rules for determining the most preferred tuples [21].

Our approach is different from existing approaches in several ways. First, we follow a hybrid implementation that is closer to the database than plug-in approaches yet not purely native as RankSQL [22] thus combining the pros of both worlds. Second, we consider preference evaluation (how preferences are evaluated on data) and filtering of preferred preferences that involve many attributes (e.g., [7]) and context-dependent preferences (e.g., [4], [26]). In the latter case, the context can be dictated by the data [4] (for example, in the context of comedies, the year of release should be after 2000) or it can be ephemeral and external to the database [26] (e.g., I like comedies when I am alone and horror films with friends).

We follow a quantitative approach that covers earlier works w.r.t. different types of preferences. In contrast to classical quantitative models (e.g., RankSQL [22]) that assume each preference being relevant for every tuple, we allow preference scores to be associated with tuples in a context-dependent way. We deal with context related to data as [4] but we follow a quantitative approach. Finally, we model the inherent uncertainty that comes with preferences, for instance, due to different learning paths, as the confidence of a preference.

Regarding preference integration and processing, we directly translate queries to conventional ones, execute them over the database, and then aggregate the results. This approach is different from previous ones, which either rewrite the query with preferences (plug-in approach) or execute the query with preferences as standard conditions (purely native approach).

In summary, the contributions of this work are the following:

- A preference-aware framework where preferences appear inside queries as first-class citizens and preference evaluation is captured as a special operator that can be combined with the classical relational operators. By separating preference evaluation from filtering preferred tuples, we can study its algebraic properties and apply them to develop holistic query processing methods.
- Flexible preference representation that concisely captures different aspects of preferences, with confidence being one that is new to this model.
- Hybrid implementation that is flexible and non-obtrusive to the database engine. Our query processing algorithms are closer to the database than plug-in approaches, yet compatible with any conventional database leveraging existing technology.
- Detailed experimental evaluation where we experimentally evaluate the efficiency and scalability of our query processing strategies. We also compare them to a plug-in strategy and we show the advantages of our approach.

II. RELATED WORK

We discuss prior work with respect to: how preferences are represented in the context of relational data and how they are integrated and processed inside relational queries.

In representing preferences, there are two approaches. In the qualitative approach, preferences are specified using binary predicates called preference relations [7], [16]. For example, a preference relation would specify that value $a$ is preferred over $b$ and $c$. In quantitative approaches, preferences are expressed as scores assigned to tuples [5], [22] or query conditions [19]. A tuple $r$ is preferred over tuple $r'$ iff the score of $r$ is higher than the score of $r'$. Existing works have studied various types of preferences following a qualitative or quantitative approach including likes and dislikes (e.g., [16], [19]), multi-granular preferences that involve many attributes (e.g., [7]) and context-dependent preferences (e.g., [4], [26]). In the latter case, the context can be dictated by the data [4] (for example, in the context of comedies, the year of release should be after 2000) or it can be ephemeral and external to the database [26] (e.g., I like comedies when I am alone and horror films with friends).
tuples (selecting preferred tuples that will appear in the result) as two operations whereas prior approaches consider them as one. Specifically, winnow [7] selects the set of all tuples not dominated by any other tuple as the set of most preferred tuples and rank [22] selects the ranked set of top-k results. In contrast, we particularly focus on preference evaluation as a single operator that can be combined with other operators and we study its algebraic properties and implementation. As a result, we develop generic query processing techniques that can be applied to queries irrespective of tuple filtering whereas prior work has developed techniques focusing on particular types of queries (for instance, for returning the top-k preferred tuples [22] or the not-dominated ones [7]).

Finally, FlexRecs [18] extends relational algebra with specialized operators for declaratively describing recommendation tuples [22] or the not-dominated ones [7].

We consider that preferences are expressed for tuples of a relational database. A database is a set of relations. A relation \( R \) is related to preference operators, including our prefer operator. Specialized operators for declaratively describing recommendation tuples [22] or the not-dominated ones [7].

In contrast, we particularly focus on preference evaluation as a single operator that can be combined with other operators and we study its algebraic properties and implementation. As a result, we develop generic query processing techniques that can be applied to queries irrespective of tuple filtering whereas prior work has developed techniques focusing on particular types of queries (for instance, for returning the top-k preferred tuples [22] or the not-dominated ones [7]).

We consider that preferences are expressed for tuples of a relational database. A database is a set of relations. A relation \( R \) has a set \( A = \{A_1, ..., A_d\} \) of attributes, and \( r \) is a tuple in the instance of \( R \). For an attribute \( A_i \), \( 1 \leq i \leq d \), we will use \( r.A_i \) to denote the value of \( r \) for \( A_i \), and \( \text{dom}(A_i) \) for the domain of \( A_i \). A preference \( p \) is defined on relation \( R \) as a triple with a conditional part that describes the tuples that are affected by the preference, a scoring part that scores these tuples, and a confidence part as a degree of certainty for the preference. More formally:

**Definition 1:** (Preference). A preference \( p \) for a set of tuples of a relation \( R \) is expressed as a triple \( (\sigma_\phi, S, C) \), where \( \sigma_\phi \) involves a set \( A_\phi \) of attributes, \( A_\phi \subseteq A \), \( S \) is a function: \( \text{dom}(A_\phi) \rightarrow [0, 1] \cup \{\bot\} \), where \( A_\phi \subseteq A \), and \( C \) is a constant in \( [0, 1] \).

The meaning of preference \( p \) is that each \( r \in \sigma_\phi(R) \) is associated with a score through the function \( S \) with confidence \( C \). A tuple \( r \) is preferred over a tuple \( r' \) if \( r \) has a higher score than \( r' \). Score \( \bot \) suggests lack of knowledge about how interesting a tuple is and it is typically the default score for a tuple. Further, higher score confidences indicate more or better evidence for a preference. For instance, if a preference \( p \) is explicitly provided by a user, we will assign the maximum confidence value, i.e., 1, to \( p \).

In general, preferences may be collected through different learning paths. For example, they may be implied based on user clicks, purchases or query logs. Confidence essentially captures the uncertainty imposed by the preference learning method. As we will see later, confidence is useful (i) as a weight that influences the total score of a database tuple (Section IV-A) and also (ii) for filtering out tuples based on credibility (Section V). For instance, we may want to return tuples that are highly likely (based on confidence) to be liked a lot (based on score). Or we may want to present some serendipitous results that may be liked (lower confidence).

In the following, we will consider two types of preferences: (i) atomic and (ii) generic. Atomic preferences are essentially user ratings and they involve exactly one tuple. On the other hand, generic preferences are usually derived automatically by the system and they are set-oriented, i.e., they are bound to a set of tuples that satisfy the conditional part of the preference.

**Example 1:** Consider the movie database depicted in Fig. 1 and let the movie ‘Million Dollar Baby’ have id \( m_3 \) in our database while ‘Gran Torino’ has id \( m_1 \). Further, assume that a user named Alice has rated both films with 8/10 and 3/10 respectively. Preference \( p_1 \) for the movie ‘Million Dollar Baby’ can be expressed as \( p_1[Movies] = (\sigma_{m_id=m_3}, 0.8, 1) \) while \( p_2[Movies] = (\sigma_{m_id=m_1}, 0.3, 1) \) captures the preference for ‘Gran Torino’. Observe that in both preferences the confidence value is set to 1; since the preferences are directly provided by users we are certain about their scores.

Now assume that the system has extracted a set of (generic) preferences for Alice, illustrated in Fig. 2, based on her search history and ratings. Since these preferences have not been explicitly expressed by Alice, the confidence value of each preference depends on how the preference has been extracted, and it will be typically lower than 1.

**Example 2:** Preference \( p_3 \) can be expressed as \( p_3[Genres] = (\sigma_{\text{genre}='Comedy'}, 1, 0.8) \). According to \( p_3 \), all movies for which \( \sigma_{\text{genre}='Comedy'} \) holds should have the maximum score.

Function \( S \) may assign the same constant value to all tuples in \( \sigma_\phi(R) \) (as we saw in the earlier example) or assign different scores to these tuples based on the values of the attributes in \( A_\phi \). There exists a considerable body of work on providing efficient methods to learn a scoring function \( S \) that ‘best’ captures a user’s preferences, for example based on user clickthrough data [14], [15], query logs [12] or by employing user feedback [8]. However, since preference learning is not the focus of this work, we will assume that appropriate scoring functions for the corresponding preference conditions have already been extracted for each user.

In what follows, we describe additional preferences to illustrate how flexible our preference formulation is in capturing elaborated preferences. For ease of presentation, in these upcoming examples, we will consider the following functions:

- \( S_{\text{atomic}}(\text{rating}) = 0.1 \cdot \text{rating} \)
- \( S_{\text{score}}(year, x) = \text{year}/x \)
- \( S_d(\text{duration}, x) = 1 - |\text{duration} - x|/x \)

**Example 3:** Let us consider preference \( p_4 \) (Fig. 2), ex-
pressed as \( p[RATINGS] = (σ_{\text{votes}>10}, S(rating), 0.8) \). For this preference, the scoring function \( S_r \) is used with input parameter a movie’s rating, which takes values in \([0, 10]\), to assign higher scores to movies with higher ratings.

The multi-attribute preference \( p_M[MOVIES] = (0.5 \cdot S_m(\text{year}, 2011) + 0.5 \cdot S_d(\text{duration}, 120), 0.9) \) uses the scoring functions \( S_m \) and \( S_d \) to score tuples based on both their year of production and duration. \( S_m \) provides higher scores to recent movies and \( S_d \) to movies with duration around 2 hours.

Generic preferences can be defined on product relations as well (i.e., multi-relational preferences). Preference \( p_6 \) in Fig. 2 is defined on the product relation \( MOVIES \Join GENRES \) as follows: \( p_6[MOVIES \Join GENRES] = (σ_{\text{genre}='Action'}, S_m(\text{year}, 2011), 0.8) \). Finally, \( p_7 \) is a membership preference, i.e., it defines a preference for tuples having a join in another relation and can be expressed as: \( p_7[MOVIES \Join AWARDS] = (σ_{\text{true}}, 1, 0.9) \).

### IV. Extended Relational Model

#### A. p-Relations

In order to enhance database tuples with the notion of preference-aware score and confidence, we introduce p-relations.

**Definition 2:** (p-relation). Given a relation \( R(A_1, A_2, ..., A_d) \), a p-relation \( R_P \) is defined by extending \( R \) with (i) a score attribute \( S \), and (ii) a confidence attribute \( C \). The default value for \( S \) is set to \( \perp \) and for \( C \) is set to 0, i.e.: \( R_P = \{ (r, S, C) | r \in R, S = \perp, C = 0 \} \).

\( r.S \) and \( r.C \) denote the score and confidence of a tuple \( r \) resp.

Notice that although the maximum score or confidence value that can be assigned by one preference is 1, tuples can have larger score or confidence values as a result of applying more than one preference and combining individual scores. In the following, whenever we use \( R \), we will refer to the corresponding p-relation without discrimination.

In general, tuple scores and confidences can be assigned either by: (i) evaluating user preferences (as we will see in Section IV-C) or (ii) combining scores/confidences available for the same tuple (e.g., when joining p-relations as we discuss in Section IV-B). In these cases, when more than one score-confidence pair for a tuple may exist, we need to combine them into a final score and confidence. For this purpose, we will use an aggregate function. Since a preference score always comes with a confidence, we will use the notation \( (S, C) \) to denote that score \( S \) has confidence \( C \). Aggregate functions need to handle both score and confidence in a unified way.

**Definition 3:** (Aggregate Function). An aggregate function \( F : (S, C) \times (S, C) \rightarrow (S, C) \) combines two confidence-score pairs and produces a new confidence-score pair.

The choice of the aggregate function to be used reflects the philosophy of how to combine partial scores into a total score and depends on the intuition of how the application should behave in the real world. In the following we give some examples:

**Example 4:** When aggregating partial scores on a tuple, we may want to incorporate the confidence of each score into the final tuple score so that scores with lower confidence will contribute less. In addition, the final confidence will reflect the total credibility of partial scores. Following this intuition, \( F_S \) computes a new score-confidence pair where the score is equal to the weighted sum of initial scores using their confidence values as weights, and the new confidence is equal to the sum of input confidence values:

\[
F_S((S_1, C_1), (S_2, C_2)) = \left\{ \begin{array}{ll}
\sum_{k \neq \perp} C_k S_k & \text{if } \exists S_k \neq \perp \\
(\perp, 0) & \text{else}
\end{array} \right.
\]

As an alternative we could also take the average of scores or confidences (or any combination). Sum better captures how the differences in scores assigned by contradictory preferences. Clearly, which strategy is more appropriate largely depends on the specific application scenario.

**Example 5:** Following a different philosophy we could argue that the tuple score should be determined by the preference with the highest confidence. In this case, we would use \( F_{\text{max}} \), which takes two pairs \( (S_1, C_1) \) and \( (S_2, C_2) \) as input and outputs the pair with the maximum confidence:

\[
F_{\text{max}}((S_1, C_1), (S_2, C_2)) = \left\{ \begin{array}{ll}
(S_k, C_k) & \text{if } \exists k \neq \perp \\
(\perp, 0) & \text{else}
\end{array} \right.
\]

Intuitively, changing the order in which preferences are evaluated should not change the final score and confidence. Therefore we expect aggregate functions to be associative. Further, aggregate functions must satisfy the following properties: (i) \( F((\perp, 0), (\perp, 0)) = (\perp, 0) \), and (ii) \( F((\perp, 0), (S, C)) = (S, C) \), i.e. \( (\perp, 0) \) is the identity element for \( F \).

It is rather trivial to prove that both \( F_S \) and \( F_{\text{max}} \) are associative; therefore proofs are omitted due to lack of space.

For the sake of simplicity (and without loss of generality), in the following we will assume that \( F_S \) is used.

#### B. Base Operators

We extend the traditional relational operators so that they can handle p-relations.

- **Select**, \( \sigma_\phi(R) \), selects tuples from relation \( R \), for which condition \( \phi \) holds, i.e.,

\[
\sigma_\phi(R) = \{ r | r \in R \text{ and } r \text{ satisfies } \phi \}
\]

- **Project**, \( \pi_{A_1, A_2, ..., A_k}(R) \), creates a new p-relation by projecting \( R \) into a smaller set of its attributes, i.e.,

\[
\pi_{A_1, A_2, ..., A_k}(R) = \{ (r.A_1, r.A_2, ..., r.A_k, r.S, r.C) | r \in R \}
\]

Note that, in addition to the projected attributes, the operator preserves the score and confidence attributes, so that the resulting relation is a p-relation.

- **Intersection**, \( R_i \bigcap_p R_j \), of p-relations \( R_i \) and \( R_j \) is a p-relation that includes all tuples that are both in \( R_i \) and \( R_j \) with preference scores (confidences) derived from the scores (confidences) of the tuples in \( R_i \) and \( R_j \), i.e.,

\[
R_i \bigcap_p R_j = \{ (r, S, C) | (r, S_i, C_i) \in R_i \land (r, S_j, C_j) \in R_j, \langle S, C \rangle := F((S_i, C_i), (S_j, C_j)) \}
\]
Union, \( R_1 \cup_p R_2 \), of p-relations \( R_1 \) and \( R_2 \) is a p-relation that includes all tuples that are either in \( R_1 \) or in \( R_2 \) or in both \( R_1 \) and \( R_2 \). Duplicate tuples are eliminated.

\[
R_1 \cup_p R_2 = \{ (r, S, C) | (r, S, C) \in R_1 \lor (r, S, C) \in R_2 \}
\]

Example 6: Assume \( R_1 \) and \( R_2 \) are two sets of movies, one for Alice and one for Bob, and say we want to find movies that they could see jointly. We use \( R_1 \cap_{\phi, F} R_2 \) to find all common movies in both input relations with scores (confidences) derived from the scores (confidences) of these movies in \( R_1 \) and \( R_2 \) using \( F_\phi \).

We now extend inner joins on p-relations:

- Inner Join, \( R_1 \bowtie_{\phi, F} R_2 \), of p-relations \( R_1 \) and \( R_2 \) is a p-relation such that:

\[
R_1 \bowtie_{\phi, F} R_2 = \{ (r, S, C) | r = r_1 \bowtie_\phi r_2, r_1 \bowtie_\phi r_2 \in R_1, (r_1, S, r_2, C) \in F_\phi \}
\]

Example 7: Consider p-relations MOVIES and DIRECTORS depicted in Figures 3(a) and 3(b). Fig. 3(c) shows the result of MOVIES \( \bowtie \) DIRECTORS.

Observe that in set operations and joins an aggregate function \( F \) is used for combining the scores a tuple has in the input relations. For simplicity we will just write \( R_1 \bowtie R_2 \) to denote \( R_1 \bowtie F_\phi R_2 \) with \( \phi \in \{\bowtleq, \bowlt \} \).

C. Prefer Operator

Standard relational algebra does not directly support evaluating preferences on p-relations. Therefore, we introduce a special prefer operator \( \lambda_{p,F}(R) \). Prefer evaluates a preference \( p \) on \( R \) using the specified aggregate function \( F \). Formally, prefer, \( \lambda_{p,F}(R) \) of a p-relation \( R \) based on \( p := (\sigma_{\phi}, S, C) \) is a p-relation such that:

\[
\lambda_{p,F}(R) = \{ (r, S', C') | (r, S, C) \in R, \text{with } S' = F((S, C), (S'(r), C'(r)), \text{if } r \in \sigma_\phi(R) \}
\]

Hereafter, we will use \( \lambda_p \) instead of \( \lambda_{p,F} \) for simplicity.

Example 8: Consider the p-relation MOVIES in Fig. 3(a) and two preferences: \( p_1[MOVIES] = (\sigma_{\text{year} \geq 2000}, S_m(\text{year}, 2011), 1) \) and \( p_2[MOVIES] = (\sigma_{\text{duration} \geq 120}, S_d(\text{duration}, 120), 0.5) \). Figures 4(a), 4(b) show the resulting p-relations after evaluating \( \lambda_{p_1,F}(MOVIES) \) and \( \lambda_{p_2}(\lambda_{p_1}(MOVIES)) \) respectively.

In the following, we present some interesting algebraic properties involving the prefer operator. These properties will be used during query optimization in order to obtain more efficient query plans as we will see in Section VI-A. All properties can be easily proved; proofs are omitted due to space considerations.

Property 4.1: If the selection condition \( \sigma_\phi \) involves only attributes other than score or confidence, then prefer and select operators commute, i.e., \( \sigma_\phi \lambda_p(R) = \lambda_p \sigma_\phi(R) \)

Property 4.2: Let preference \( p[R] = (\sigma_\phi, S, C) \) and a selection condition \( \sigma_\phi \) involving only attributes other than score and confidence. Then \( \sigma_\phi \lambda_p(R) = \sigma_\phi \lambda_p(R) \) where \( p[R'] = (\sigma_\phi \land \phi', S, C) \)

Recall that (i) \( F \) is associative and commutative, and (ii) the same \( F \) is used across all operations. Then the following properties hold:

Property 4.3: Prefer is commutative, i.e., \( \lambda_{p_1}(\lambda_{p_2}(R)) = \lambda_{p_2}(\lambda_{p_1}(R)) \)

In other words, changing the order of preference evaluation will not change the final score and confidence values.

Furthermore, the extended base operators defined in Section IV-B can be combined with the prefer operator. For instance, the prefer operator can be pushed over a binary operator \( \theta \in \{\bowtleq, \bowlt \} \). In the case that preference \( p \) is defined solely using attributes of \( R_i \), then evaluating \( p \) on a different relation \( R_j \) will result to \( \lambda_p(R_j) = R_j \), i.e., the score and confidence values of \( R_j \) will remain unchanged. Thereby, we get the following property:

Property 4.4: Given a preference \( p \) defined exclusively using attributes of \( R_i \), then it holds \( \lambda_p(R_i \bowtleq_j R_j) = \lambda_p(R_i) \bowtleq_j R_j \)

V. PREFERENTIAL QUERIES

The prefer operator and the extended relational operators comprise the core of our preference-aware relational algebra. In this section we present our query model that is based on this
extended relational algebra. We will first describe the query model that we follow; subsequently we provide examples that showcase the expressivity and flexibility of queries expressed using our extended relational algebra.

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.

Intuitively, 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.

Users are not expected to directly formulate preferential queries. Rather, we will assume that preference-aware applications will provide an appropriate interface that facilitates users to specify their searching criteria and collected preferences are automatically integrated into their queries, (e.g. based on techniques such as those proposed in [19] or [4]). Consequently, users are not exposed to the details, similarly to a typical database-driven application where users are not required to specify queries expressed in SQL.

As an application scenario consider an online video rental service that uses the database of Fig. 1. The user interface supports searching over the available movie titles, browsing reviews, getting recommendations, etc. The system collects user preferences based on their search history, rentals and reviews, getting recommendations, etc. The system collects

In this section, we describe the components of our prototype system and the implementation of p-relations and the operators. The following subsections focus on our query evaluation algorithms.

**System Architecture.** Our prototype system depicted in Fig. 6, consists of the following modules:

- **The query parser** takes as input the user query and related preferences and generates a preferential query expressed in the extended relational algebra. Further, it adds projections for all attributes that will be used as part of a prefer operator and for all join attributes as well. Finally, the query parser generates a baseline *extended query plan* keeping the order of the operators as defined in the query. An extended query plan is an expression tree whose leaves are p-relations and internal nodes are extended relational and prefer operators. Fig. 7(a) illustrates an extended query plan.

- **The query optimizer** rewrites the input query plan into a more efficient one based on the algebraic properties of the prefer operator (as discussed in Section IV-C).
The execution engine receives from the query optimizer the optimized query plan and realizes the execution of the plan.

Our main design goal is to get a proof of concept for the proposed framework that would lead to more advanced query optimizations while maintaining compatibility with any standard DBMS. Therefore, we have followed a hybrid approach in our implementation. Our prototype system works closer to the database engine than plug-in approaches enabling operator-level query optimizations (Section VI-A) while at the same time has a minimum impact on the query engine, and its query execution strategies leverage several optimizations available by the native database engine (Section VI-B).

Implementing p-relations. In implementing p-relations, one option is to store two more attributes per relation, one for the score and one for the confidence. However, most tuples remain unaffected by any preference in a single query, so they would only have their default values. In addition, scores and confidences are query-dependent, therefore permanently storing them in the database does not make sense. With these issues in mind, for each relation \( R \) affected by preferences in a query, we keep a score relation \( R_P(pk, \text{score}, \text{conf}) \), where \( pk \) is the primary key of \( R \). Note that the primary key might actually be a composite key of \( R \). For example, when evaluating a join or set operation, the resulting score relation will consist of the primary keys of individual relations, along with the score and confidence attributes. While each base relation may be affected by multiple preferences in a query, we kept the prototype implementation simple by using only one score relation per base relation, which is updated with new score and confidence values each time a preference-related operation (e.g., a prefer or a join) is executed. To save more space, each score relation \( R_P \) contains only tuples with non-default scores and confidences, consequently \(|R_P| \ll |R|\).

Physical operators’ implementation. We have implemented the extended relational and prefer operators as user defined functions that are linked with the execution engine. Projections are evaluated directly on base relations. The extended select, join and set operators are executed on both \( R \) and \( R_P \). When a select operator is evaluated, tuples that do not satisfy the conditions are filtered out from both relations. Join and set operators are evaluated in two steps. First, a conventional join or set operation is evaluated on the input base relations. Then, for the tuples in the result, their scores and confidences are calculated by applying the corresponding aggregate function on the individual score relations.

Evaluating a prefer operator is more complicated. First, the conditional part of the preference is executed on both \( R \) and \( R_P \). All qualifying tuples of \( R_P \) already have non-default scores and confidences assigned, which have to be updated with new values. In addition, all qualifying tuples of \( R \) that do not appear in \( R_P \) have to be assigned new non-default scores and confidences and be added to \( R_P \). In order to calculate the new scores for both sets of tuples, the scoring part is executed on the qualifying tuples of the base relation \( R \). Then, the corresponding aggregate function is called to calculate the final scores from the previous and the new ones.

A. Query Optimizer

The query optimizer’s goal is to improve the input query plan by applying query transformation rules that leverage the algebraic properties of the prefer and the extended relational operators. Note that, since we have only one implementation for our operators, in the following we will consider only the space of logical (and not physical) execution plans.

Although estimating the cost of the alternative execution plans is quite complex, generally the most critical parameter that shapes the processing cost is the disk I/Os, which in turn depends on the size of the intermediate relations, i.e. how many tuples are materialized during the query plan execution. Therefore, a primary optimization goal for the query optimizer is to minimize the size of intermediate relations. For this purpose, our query optimizer applies the following set of heuristic transformation rules:

1. Selections are pushed down the query plan as far as they can go. If the selection condition involves several relations, the condition is split and each piece is pushed down separately.
2. Projections are pushed down the query plan as far as possible.
3. Prefer operators are pushed down the query plan, just on top of a select or project operator, whenever applicable (based on Prop. 4.1).
4. If a prefer operator defined over a binary operator (join, set operation, etc.) involves attributes only on one relation, it is pushed to the corresponding relation (based on Prop. 4.4).
5. If several prefer operators are defined on the same relation, they are ordered in ascending selectivity of their conditional parts (based on Prop. 4.3).

Heuristics 1 and 2 are commonly used in query optimization for restricting the number and size of tuples (respectively) passed through the execution plan. Similarly, Heuristic 3 aims at reducing the input size of prefer operators. Heuristic 4 is based on the expectation that it is likely for a relation to provide index-based access for the attributes used by the prefer operator. In contrast, typically the product of a join will not be indexed. Hence, evaluating a preference directly on the respective relation should be faster. Finally, heuristic 5 aims at minimizing the number of tuples produced when materializing score relations by incrementally evaluating preferences from less to more expensive.
Example 12: To illustrate these rules, consider the extended query plan depicted in Fig. 7(a) (projection operators are omitted for simplicity). Assume that both \( \sigma_\Phi \) and \( \lambda_{p_2} \) involve attributes only on relation \( R \). Based on rules 1 and 3, the two operators are pushed down to the corresponding relation \( R \). Further, assume that the conditional part of \( p_2[R] \) is more restrictive than that of \( p_1[R] \). Hence, \( \lambda_{p_2} \) and \( \lambda_{p_3} \) are reordered according to rule 5. Figure 7(b) illustrates a more efficient query plan produced by the query optimizer. □

Further, the query optimizer rearranges the subtrees of the optimized plan in order to construct a left-deep plan. The reason for preferring a left-deep plan is to save memory space, because at any time during the execution of a left-deep plan only two temporary relations have to be kept in memory. Finally, in order to further optimize the non-preference part of the query, the query optimizer rearranges the subtrees of the extended query plan in order to match the join order that would be followed by the native query optimizer.  

B. Execution Engine

The execution engine is responsible for the execution of an extended query plan and supports different query evaluation strategies, which are described next. We will first present an algorithm, termed \( FP \), that essentially separates preference evaluation from the execution of the non-preference query part. Next, we will introduce query evaluation algorithms that work on the query plan generated by the query optimizer, hence taking advantage of the preference-based query optimizations that we discussed in Section VI-A.

Filter-then-Prefer Algorithm. The basic intuition behind the Filter-then-Prefer (FP) algorithm is that if we somehow distinguish the non-preference query processing from the prefer operators’ evaluation, we could delegate the non-preference part to be executed by the database. Subsequently, we would just need to apply all the remaining prefer operators on the qualifying tuples, similarly to plug-in approaches.

The \( FP \) algorithm (Alg. 1) first extracts the non-preference query part \( Q_{NP} \) from \( Q \) by removing all prefer operators. The non-preference query part is essentially the same with the corresponding SQL query that can be constructed by not considering any prefer operator, with one exception; it contains additional projections for all attributes that will be used by any prefer operator (either for the conditional or scoring part of a preference). Recall that these additional projections have already been added by the query parser. The motivation for this modification is to enable prefer operators’ evaluation directly on the result of \( Q_{NP} \), without requiring additional joins with the respective base relations.

The execution of \( FP \) proceeds as follows. \( Q_{NP} \) is executed as usual by the query engine and let \( R_{NP} \) be the resulting relation. Then, the extended query plan is executed in a bottom-up fashion. All non-preference operators have already been evaluated, hence they are ignored. Prefer operators are evaluated on \( R_{NP} \) instead of the corresponding base relations, whereas join and set operations are evaluated only on the corresponding score relations. \( FP \) terminates when the root node is visited. Finally, a join between \( R_{NP} \) and \( R_P \) is executed to produce the query results.

Now we present two algorithms that exploit the preference-based query optimizations discussed in Section VI-A.

Bottom-Up Algorithm. The Bottom-Up (BU) algorithm performs a postorder traversal of the execution plan provided by the query optimizer and executes each operator following the order of execution defined in the input query plan. Each operation is executed by calling the corresponding user defined function. For any internal node (operator) \( n_i \) in the execution plan, the algorithm maintains the intermediate results of executing \( n_i \) as a pair of temporary relations \( R_i \) and \( R_{P_i} \). At any time during the plan execution, \( BU \) maintains only the intermediate relations required by the following operators. Assuming left-deep query plans, only two intermediate relations are required at any time. \( BU \) terminates when the root node \( n_r \) is executed; the final query results are produced by joining \( R_r \) with \( R_{P_r} \).

Group Bottom-Up Algorithm. The Bottom-Up algorithm is greedy in the sense that it directly and separately executes each operation and materializes the temporary results. Assume that several standard operators (i.e., selections, projections or joins) have to be executed sequentially without any prefer operator in between. In this case, we can combine these operators into one query that is passed to the database; hence we can leverage

---

Algorithm 1: Filter-then-Prefer

| Input: \( Q_P \) a query plan, \( n_r \) the root of \( Q_P \) |
| Output: \( R_Q = \{(r, C, S) | r \in R_{NP}\} \) where \( R_{NP} \) is the result of executing the non-preference part of \( Q \) |
| Variables: \( R_P \) a set of intermediate score relations |
| \( R_{NP} \) the answer of the non-preference part of \( Q \) |
| \( R_{P} \) a set of intermediate score relations |

```
begin
1. \( R_P := \emptyset; \)
2. extractNPQuery(\( Q_P \)) \rightarrow \( q' \);
3. execute(\( q' \)) \rightarrow \( R_{NP}; \)
4. \( FP(n_r, R_{NP}, R_P); \)
5. \( R_Q := R_r \land R_{NP}; \)
6. return \( R_Q; \)
```

Function \( FP \)

Input: \( n_i \) node in \( Q_P \), \( R_{NP} \) the answer of the non-preference part of \( Q \), \( R_P \)
Output: \( R_P \)

```
begin
1. if \( n_i \) is null then
2. exit;
3. FP(left, \( R_{NP} \), \( R_P \));
4. FP(right, \( R_{NP} \), \( R_P \));
5. if \( n_i \) is a relation, select or project operator then
6. let \( n_1 \) be the child node of \( n_i \);
7. remove \( R_{P_i} \) from \( R_P \);
8. evaluate(\( n_i, R_{NP}, R_{P_i} \)) \rightarrow \( R_{P_i}; \)
9. insert \( R_{P_i} \) into \( R_P \)
10. if \( n_i \) is a join operator then
11. let \( n_1, \ldots n_k \) be the children nodes of \( n_i \);
12. remove \( R_{P_i}, R_{P_{n_k}} \) from \( R_P \);
13. evaluate(\( n_i, R_{NP}, R_{P_i}, R_{P_{n_k}} \)) \rightarrow \( R_{P_i}; \)
14. insert \( R_{P_i} \) into \( R_P \)
```

1Note that in most standard DBMSs the optimal query plan can be retrieved without actually executing the query. For instance in PostgreSQL this functionality is supported by issuing an EXPLAIN [query] command (with negligible processing overhead).
the query processing capabilities of the native query engine. To illustrate, consider Fig. 7(c). Observe that it is possible to combine $\sigma_{a_2}$ with $T \bowtie U$ and $(T \bowtie U) \bowtie S$ into a single operation.

The Group Bottom-Up (GBU) algorithm (Alg. 2) performs a postorder traversal of the extended query tree following a strategy that saves intermediate materializations. It defers operator execution wherever possible, and combines operators that can be executed together into a single query that is delegated to the database engine.

To enable batch execution of operators, $GBU$ maintains an additional directed acyclic graph $G$ that contains the last intermediate relations produced and all operators that can be combined. During the plan traversal, whenever $GBU$ visits a node $n_i$ that is a non-preference operator, $n_i$ is copied to $G$. Whenever the algorithm visits a prefer operator node $n_i$, then all nodes belonging to the subtree of $n_i$ are removed from $G$ and combined to a single query that is passed to the query engine. Query results are materialized and given as input to $n_i$. The intermediate relation produced by evaluating $n_i$ is again inserted into $G$. When $GBU$ reaches the root of the execution plan, all remaining operators in $G$ are combined and executed. Finally, the relations produced after executing the root are joined to produce the final query results.

VII. EXPERIMENTAL EVALUATION

In this section, we provide experimental results that evaluate the performance of our methods. All experiments were conducted on a 2.0 GHz Intel Xeon CPU with 16 GB RAM running Debian Linux 2.6.24 operating system and PostgreSQL 8.4.3. The shared buffers (shared memory buffer size) and work_mem (internal memory for sorting and hashing) parameters in PostgreSQL were configured to 3 GB and 500 MB respectively. The prefer and extended relational operators as well as aggregate and scoring functions were built as user-defined functions in pgSQL (a built-in procedural language supported by PostgreSQL). The remaining modules (query parser, query optimizer and execution engine) were implemented in Java using JDBC.

A. Experimental Setup

Data sets. We used two real data sets in our performance evaluation: the IMDB [2] and the DBLP [1] data sets. The IMDB data set is a snapshot of the publicly available movie database taken in March 2010, with 500 MB total size. Fig. 1 illustrates part of the IMDB database schema used in the experiments. The DBLP data set was extracted in June 2011 and has 850 MB total size. For the DBLP database we decomposed part of the XML data set into the database schema shown in Fig. 8. The sizes of the basic tables used in the experiments are depicted in Table I.

Queriers. We constructed a workload consisting of 6 queries to be used in the experiments, which we denote IMDB-1 to IMDB-3 and DBLP-1 to DBLP-3. Table II summarizes some important properties for the examined queries. $N$ is the query result size, $|R|$ represents the number of joined relations, $|\lambda|$ is the number of preferences. Finally, $|P|/|NP|$ represents the number of relations with preferences vs. the number of relations without preferences respectively. In our experiments, we use these queries as our basic test scenarios as well as variations of them by modifying their parameters. In each set of experiments, we vary a single parameter of a query while keeping everything else fixed. We will explicitly describe which query we modify and how in each case.

Each experiment compares the efficiency of our algorithms in terms of the total query processing time, measured with the EXPLAIN ANALYZE command of PostgreSQL. In order to obtain more accurate measurements, each experiment was conducted with cold cache, i.e., we flushed the cache before running each experiment. Note that we have excluded $BU$ algorithm from our experimental evaluation, as $GBU$ is an improved method over $BU$.
Baseline Approach. We further compare the efficiency of our methods with respect to a pure plug-in strategy that evaluates preferences on top of the database, similar to the approaches taken by [17], [19]. A plug-in strategy involves the following phases: (i) Rewrite where the preferences are integrated as standard query conditions in the query producing a set of new queries, (ii) Materialize where the new queries are executed and (iii) Aggregate where the partial results are combined into a single ranked list.

In particular, we have implemented two variations of this plug-in strategy. In the first variation, namely RMA-m, a new query is constructed by integrating one preference condition as an additional filtering condition. Each query is executed and the results are scored based on the scoring function of the respective preference. Finally, the partial scores are aggregated to produce the final score and confidence values. In the second variation, namely RMA-1, first each preference condition is added to the query in disjunction with the other preference conditions. Only one (disjunctive) query is constructed. The new query is executed and the scoring function of each preference is applied to the query results. Finally, likewise RMA-m, partial scores are aggregated and final scores and confidence values are calculated.

To illustrate how these plug-in methods work, consider the following query (IMDB-1):

```
SELECT m.m_id, m.title, r.rating, g.genre
FROM MOVIES m, RATINGS r, GENRES g
WHERE m.m_id = r.m_id AND m.m_id = g.m_id AND m.year ≥ 2009
```

Assume the system has learned the following preferences for Alice: She likes comedies (p₁), she trusts user ratings (p₂) and she prefers films that last less than 2.5 hours (p₃), i.e.:

```
p₁(GENRES) = [σgenre = 'Comedy', 0.9, 0.8]
p₂(RATINGS) = [σvotes ≥ 100, Sr(rating), 0.9]
p₃(MOVIES) = [σduration < 150, Sd(duration), 1]
```

Following the RMA-m approach, the following queries would be built:

```
Q₁:SELECT m.m_id, m.title, 0.9 as score, 0.8 as conf
FROM MOVIES m, RATINGS r, GENRES g
WHERE m.m_id = r.m_id AND m.m_id = g.m_id AND m.year ≥ 2009
AND g.genre = 'Comedy'
```

```
Q₂:SELECT m.m_id, m.title, Sr(r.rating) as score, 0.9 as conf
FROM MOVIES m, RATINGS r, GENRES g
WHERE m.m_id = r.m_id AND m.m_id = g.m_id AND m.year ≥ 2009
AND m.votes ≥ 100
```

```
Q₃:SELECT m.m_id, m.title, Sd(m.duration) as score, 1 as conf
FROM MOVIES m, RATINGS r, GENRES g
WHERE m.m_id = r.m_id AND m.m_id = g.m_id AND m.year ≥ 2009
AND m.duration < 150
```

In the next step, queries Q₁-Q₃ are executed and results are materialized into tables T₁-T₃. Finally, the following query aggregates partial results into a single ranked list:

```
SELECT m.m_id, m.title, r.rating, g.genre, t.score, t.conf
FROM MOVIES m, RATINGS r, GENRES g, TEMP t AS
(SELECT m.m_id, AGGR(score) as score, AGGR(conf) as conf
FROM T₁ UNION T₂ UNION T₃
GROUP BY m_id)
WHERE m.m_id = r.m_id AND m.m_id = g.m_id AND m.m_id = t.m_id AND m.year ≥ 2009
```

where AGGR corresponds to the aggregate function used.

In contrast, the RMA-1 strategy would build a single query that would combine all preferences together in a complex qualification in the WHERE clause.

Clearly, due to the need to translate preferences into traditional query constructs, even fairly simple queries are hard to express or be optimized by the query engine. This fact justifies the motivation of this work for pushing preference evaluation closer to the database as a special operator. Further, as it will be showcased in the experiments, there is no clear rule that, given a query, would determine which plug-in method is faster.

B. Experimental Results

Sensitivity vs. |R|. In the first experiment, we measured the performance of all methods with respect to the number of relations |R|. In particular, we used IMDB-1 with |R| = 3 as a test query and gradually joined more relations, up to |R| = 6. Fig. 9(a) shows the measured query processing time against the number of relations. As expected, the performance of all algorithms deteriorates with the number of joins. Further, both FP and GBU are significantly faster and scale better with the number of joined relations than the plug-in implementations.

Sensitivity vs. N. Next we examined the processing cost of all algorithms with respect to the cardinality of the input relations. In particular, we generated subsets of the input relations with different cardinalities by randomly selecting a specific number of tuples from one relation and keeping only joined tuples from all the other relations. For example for the IMDB database, we randomly selected 1M tuples from the MOVIES relation, then 500K from them, and so on. Figures 9(b) - 9(d) illustrate the processing cost of all algorithms for queries IMDB-1 - IMDB-2 and DBLP-1, where the x-axis represents the cardinality of the basic relation (i.e. MOVIES for the IMDB database and PUBLICATIONS for the DBLP database respectively). As depicted, GBU algorithm is the most efficient execution strategy regardless of the size of input relations.

Sensitivity vs. |λ|. In this experimental setting, we investigate the performance of all methods with respect to the number of preferences that have to be evaluated on each relation. In particular, for every relation on which a preference is defined, we varied the number |λ| of prefer operators to be evaluated from 1 to 5. Figures 10(a) - 10(f) show the results for queries IMDB-1 - IMDB-3 and DBLP-1 - DBLP-3 respectively.
With respect to the plug-in approaches, recall that they translate preferences into additional query constructs. As the number of preferences that have to be evaluated increases, the number of new queries to be executed grows (in the case of RMA-m) or the new query becomes more expensive and hard to optimize (in the case of RMA-1). As illustrated, for both methods the processing cost increases dramatically and it is significantly higher than the cost of our algorithms, especially in queries IMDB-1 - IMDB-3 (Figures 10(a) - 10(c)).

Further, as depicted in Figures 10(a) and 10(b), GBU is faster than FP for the less expensive queries IMDB-1 and IMDB-2 that we examined (refer to Table II). However, notice that as we move towards queries that produce more results, the difference in performance between GBU and FP is smoothed. For the most expensive queries (DBLP-2 and DBLP-3) FP outperforms GBU. To understand why this happens, recall that the processing cost depends on the size of intermediate results that are materialized during the query execution. For this experiment the non-preference query part is fixed; therefore the difference is due to the sizes of score relations. The more preferences that have to be evaluated, and the more expensive a query is, the more tuples will be passed to subsequent operators. GBU will typically evaluate preferences on the respective base relations, hence requiring score relations to be materialized several times up to the root of the execution plan. In contrast, FP will execute all prefer operators at the end, thus avoiding some intermediate materializations.

**Sensitivity vs. |P|/[NP].** Next, for each query that we examined we vary the number of relations on which prefer operators have to be evaluated. Specifically, we kept the non-preference (SQL) part of each query fixed and measured the processing cost when prefer operators are applied to 1 up to |R| out of the |R| relations that comprise the query. Figures 11(a) - 11(b) plot the measured processing time for queries IMDB-1 - IMDB-2 against |P|/[NP].

Since the non-preference part of the queries is fixed, the performance of FP depends on the preference evaluation cost, which increases with the number of preferences. On the other hand, varying |P|/[NP] affects the query plan followed by GBU. Specifically, when prefer operators are defined only on few, specific relations, then GBU leverages the optimizations that the native query optimizer provides. However, as more relations are affected by preference evaluation, GBU essentially turns into a bottom-up traversal of the query plan, and hence its performance degrades.

**Sensitivity vs. Execution Plan.** Finally, we studied the impact of selecting different execution plans on the processing cost. Since the plug-in methods do not follow the extended query plan, their study is irrelevant from this set of experiments. Further, the cost of FP algorithm is primarily determined by the non-preference query part. Thus we have excluded both plug-in and FP strategies from this experiment. For our study we considered two alternative execution plans, denoted as P_1 and P_2 respectively. Both plans follow the same join order on the non-preference operators that the native query
optimizer would select; their difference is in the positions of prefer operators inside the plan. In particular, in $P_1$ all prefer operators are evaluated at the end of the query plan, whereas in $P_2$ each prefer operator is directly evaluated on the corresponding relation.

Figures 12(a)-12(b) show the results for the tested queries. As illustrated, (with the exception of IMDB-3) in most of the examined queries $P_1$ outperforms $P_2$. Clearly, selecting a cheap execution plan could improve the performance of the GBU algorithm, which justifies our motivation for preference-based query optimizations. Further, our results provide a good starting point for exploring more sophisticated, cost-based optimizations for GBU, which is part of our ongoing research.

**Summary.** Concluding, we can make the following observations. Plug-in approaches are less efficient and less scalable than our preference-aware strategies in most of the experimental settings that we examined. Our experiments show that GBU is generally less sensitive than FP with respect to the sizes of input relations $N$ and the number of joins $|R|$ (Figures 9(a)-9(d)). The main factor that determines the performance of the GBU algorithm is the cardinality of the query output, which affects the size of intermediate score relations that are materialized during the query execution. For less expensive queries such as IMDB-1 and IMDB-2, GBU is faster than FP. The two algorithms exhibit similar performance for mid-sized queries (IMDB-3 and DBLP-1), whereas for the more expensive queries, such as DBLP-2 and DBLP-3, FP is the most efficient. Further, our experiments (Figures 12(a)-12(b)) indicate that selecting a cheap execution plan both in terms of the join order and the position of prefer operators inside the plan could significantly reduce the processing cost of GBU.

As a general observation, based on our experimental findings, blending preference evaluation with traditional relational operators might significantly improve the performance of preferential query processing. Our experimental results provide a good motivation for investigating more sophisticated, cost-based optimizations that take into account the properties of the query, data and available preferences.

**VIII. Conclusion and Future Work**

In this work we introduced a preference-aware data model where preferences appear as first-class citizens and preference evaluation is captured as a special ‘prefer’ operator. We studied the algebraic properties of the new operator and applied them in order to develop more fine-grained query optimizations and holistic query processing methods. We presented a preference-aware framework that is (i) flexible in handling different flavors of preferential queries, (ii) closer to the database than plug-in approaches, (iii) yet non-obtrusive to the database engine. Our detailed experimental evaluation using a prototype implementation showed the potential of our approach in comparison with two variations of the plug-in strategy.

In the future, we aim to develop cost-based optimizations of the prefer operator and explore the possibilities of combining the prefer operator with the rank and rank join operators defined in [13], [22] in order to enable early pruning of results based on score or confidence during query processing.

**References**