Personalized DBMS: an Elephant in Disguise or a Chameleon?

Georgia Koutrika
IBM Almaden Research Center, USA

Abstract

A query, expressed in a structured query language such as SQL, describes precisely the data that should comprise the answer and a non-empty answer is returned only if it satisfies all query criteria. However, the Boolean database query model has limitations. A user may run into empty answers when the query conditions are too restrictive or sift through too many results that match the query. User preferences naturally blend into queries allowing to explore alternatives or prioritize and filter available choices. In addition, they can be used to generate answers that are personalized to the user. In this paper, we explain the need towards personalized and preference-aware query answering in the context of databases and we discuss how it can be enabled. Our objective is to discuss the pros and cons of existing approaches and the challenges and opportunities towards a truly personalized preference-aware DBMS.

1 Introduction

Database systems answer queries accurately and deterministically. A query, expressed in a structured query language such as SQL, describes precisely the data that should comprise the answer and a non-empty answer is returned only if it satisfies all query criteria. The DBMS guarantees that the same exact answer will be generated every time the query is issued as long as the data does not change. This exact-match query paradigm serves well many scenarios. When querying your bank account balance, you always expect to get the exact amount and this amount should be the same every time a check balance request is issued as long as no deposits or withdrawals have taken place. Or when a credit card company is preparing customer bills, it is important that the correct amounts are computed and displayed on each bill.

However, the database query model may sometimes be very strict. For example, when accessing databases on the web, whose schema and contents are unknown, it may be difficult to formulate accurate queries. In this context, a user may run into either of two problems: (*i*) the empty-answer problem when the query conditions are too restrictive or (*ii*) the too-many-answers problem when too many results match the query. Furthermore, information abundance and user heterogeneity are additional factors that make the exact-match query paradigm often not appropriate. Due to information abundance, showing all possible information that matches a query may be overwhelming for the user. User heterogeneity means that different users may find different things relevant when searching because of different preferences and goals [17]. Especially on the web, it is critical to provide the 'right' answer to a user rather than the exact same answer to everyone for the same query.

Introducing preferences in query answering can help cope with the issues mentioned above. The empty-answer problem can be tackled by relaxing some of the hard constraints in the query, i.e., considering them as

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Movies

movie_id	title	year	genre
m_1	Anger Management	2003	comedy
m_2	The Proposal	2009	comedy
m_3	The Odd couple	1968	comedy
m_4	New in Town	2009	comedy

Actors

actor_id	movie_id	name
a_1	m_1	Adam Sandler
a_2	m_1	Jack Nicholson
a_3	m_2	Sandra Bullock
a_4	m_3	Walter Matau
a_5	m_4	Renée Zellweger

Figure 1: An example database instance

soft or as user wishes or by replacing them by constraints that capture user preferences related to the query. The too-many-answers problem can be tackled by strengthening the query with additional conditions expressing user preferences to restrict the results. In both cases, results can be prioritized according to how well they match the preferences in the query.

Furthermore, incorporating user preferences in query answering can help generate answers customized to each user. To illustrate this personalized answer paradigm, let us consider a web-based movie database. Figure 1 shows a snapshot containing new arrivals. Two users, Bob and Diana, issue the following queries on: (Q_1) comedies released after 2000; (Q_2) movies with Ben Stiller; and (Q_3) new comedies. Based on the exact-match database query paradigm, the same answers will be returned to both users, which are the following: (*i*) $\{m_1, m_2, m_4\}$ for Q_1, (*ii*) $\{\}$ for Q_2 and (*iii*) all new arrivals for Q_3. On the other hand, a personalized database query model would possibly generate a different answer for each query depending on the targeted user. For instance, for Q_1, Bob could receive $\{m_1, m_2\}$ because he is not a fan of actress R. Zelweger. For Q_2, showing movies that are a close match to the movies described by the query would potentially be better than an empty answer. For example, for Diana who also likes Adam Sandler and has liked movies where both Ben Stiller and Adam Sandler have co-starred, m_1 might be suggested as an alternative answer. Finally, results for Q_3 can be ordered according to a user's preferences so that each individual can easily sift through them.

Preferences naturally relax the exact-match query model by allowing to prioritize and filter available choices (as in Q_1 and Q_3) or to explore alternatives (as in Q_2). Moreover, embedding preferences in queries enable a shift from 'consensus relevancy', where the computed relevancy for the entire population is presumed relevant for each user, towards 'personal relevancy', where relevancy is computed based on each individual's characteristics.

In this article, we review related work on *preference-aware query answering* in databases. We discuss (*i*) how preferences are represented in the context of database queries, (*ii*) how the answer of a query with preferences is defined, and (*iii*) how preference-aware query answering is implemented. This review aims at showing how tightly preferences are currently coupled with database queries and sharing our vision regarding opportunities and challenges in fully implementing preferences as first-class citizens inside the database engine by changing both the database query model and internal code. Finally, this article highlights the challenges in going from preference-aware query answering to *personalized preference-aware query answering*, where the database automatically customizes answers taking into account user preferences.

2 Preference Representations

Understanding preferences and finding appropriate representations for them is a real challenge. Preferences have been studied for several years in fields such as philosophy (e.g., [8]), psychology (e.g., [18]) and economics (e.g., [6]). In recent years, preference modeling has drawn the attention of researchers from computer science fields including artificial intelligence (e.g., [22]) and databases.

In databases, preference representation approaches (preference models) are divided into two categories. *Qualitative preferences* are specified using binary preference relations that relatively compare tuples. Preference relations may be specified using logical formulas [4] or special preference constructors [11]. *Quantitative preferences* are expressed by assigning scores to particular tuples [1] or query conditions that describe sets of

tuples (e.g., [14]), where a score expresses degree of interest.

To illustrate, let us consider the database schema of Figure 1 and assume that t denotes a tuple and $t[A]$ stands for the projection of t on one of its attributes, A. A preference for recent movies can be expressed in a qualitative way as follows: movie t_i is preferred over movie t_j iff $t_i[year] > t_j[year]$. This formulation is natural for humans and results in a partial ordering of the query results. Absolute specification of the significance of a preference is not possible. On the other hand, using a scoring function such as $f(t) = 0.001 \times t[year]$, the same preference can be captured in a quantitative way. A movie t_i is preferred over movie t_j if the score of t_i is higher than the score of t_j. This formulation leads to a total ordering of the results based on user preferences.

Existing works have studied various types of preferences following a qualitative or quantitative approach including likes and dislikes [11, 14], context-dependent preferences [29], set preferences [23], and multi-granular preferences [13]. A survey on preference models in the context of databases can be found in [19].

3 Integrating Preferences into Queries

In the exact-match database world, the answer to a query is unanimously defined. Interestingly, there seems to be no single definition regarding how preferences should be interpreted in the context of database queries.

One reason is that existing approaches have thought of preferences as a means of solving seemingly different problems. Driven by the empty-answer problem, several approaches treat preferences as soft constraints that are part of the query [4, 11]. In contrast to the classical hard constraints of the query, soft constraints may be optionally satisfied in the final answer. Dealing with the too-many-answers problem, other approaches view preferences as additional constraints that can be applied in conjunction with the query conditions to prioritize and possibly restrict the query results [14, 29]. However, more importantly, the lack of a single preference-aware query model is due to the fact that preferences can be defined either qualitatively or quantitatively and different approaches for embedding qualitative or quantitative preferences into queries have emerged.

If preferences are expressed in a qualitative way (i.e., using preference relations), then they are embedded into relational query languages through an operator that selects from its argument relation the set of the most preferred tuples according to a given preference relation. This operator has been given different names such as winnow [4], BMO [11] and skyline [2]. The set of most preferred tuples is typically defined as the set of all tuples not dominated by any other tuple. Many variations of this definition have been proposed mainly due to the computational complexity of the optimal answer based on the original definition. For instance, according to the k-dominant skyline definition, a tuple t_i k-dominates another tuple t_j if there are k dimensions, or preferences, in which t_i is better than or equal to t_j and t_i is better in at least one of these k dimensions [3]. Ranking the whole input can be achieved by repetitive applications of this preference operator.

On the other hand, if preferences are expressed quantitatively (i.e., using scores) and there is an aggregating function for combining the partial scoring predicates, then tuples are assigned scores, and the answer of a query with preferences is defined as the ranked set of top-n results [5, 9]. A complementary approach is to make sure that the query results satisfy at least some of the specified preferences [14].

There has also been a significant number of proposals for extending SQL with special preference operators. For example, skylines were first introduced in SQL using the following clause [2]:

SKYLINE OF A_1 [MIN | MAX | DIFF], ..., A_m [MIN | MAX | DIFF]

where a MIN (or MAX) specification following an attribute A_i expresses preference of small (or large) values of A_i whereas DIFF expresses that only tuples with identical values of A_i are comparable.

The top-n preference operator was introduced in SQL by adding a clause of the form:

ORDER BY [user-defined-function] STOP AFTER n

In this case, the tuples in the result are ranked according to a user-defined scoring function and the results with the n highest scores are returned.

As yet another example, Preference SQL supports soft constraints using a special PREFERRING clause [11]:

PREFERRING [conditions] BUT ONLY [conditions]

Preference SQL follows a "best matches only" (BMO) semantics. First, it finds all perfect matches to the conditions of PREFERRING. If this set is empty, then it considers all other best matching values excluding those that do not satisfy the BUT ONLY conditions.

As a result of the proliferation of different interpretations of preferences in the context of database queries, no single preference-aware query model has been widely adopted in the database world and implemented as an alternative or an extension of the exact-match database query model. As a first step towards reaching consensus, we need to understand that we do not try to solve two disconnected problems by integrating preferences into queries: both the empty-answer and the too-many-answers problems are two sides of the same coin caused by the strictness of the exact-match query model. One can go from an empty answer to too many results and vice versa. For example, integrating preferences as additional constraints to restrict a query with too many answers may lead to an empty answer. Bringing preferences into queries is a means to 'relax' the exact-match query model and tackle both these problems in a unified way.

More importantly, a widely accepted preference-aware query model needs to transparently extend the exact-match model with the notion of preferences. One challenge in designing such a model is to come up with new query components that capture the main concepts of preference-aware query answering without adding too much complexity. Ideally, these components should be oblivious to how preferences are represented or implemented and they should naturally blend with the traditional query constructs. In addition, a preference-aware query model should be flexible regarding how preferences are treated within a query. For instance, all preferences could be considered optional or a lower bound on the number of preferences that should be satisfied in the results could be provided by the user.

At a high level, we envision that a preference-aware query would contain the following parts:

- *hard constraints* that are exactly matched
- *soft constraints* (i.e., preferences) that are optionally satisfied. If a query has no soft constraints then we roll back to the exact-match query paradigm, where the answer satisfies exactly the hard conditions.
- a *preference bound* that specifies how preferences should be treated: an *any preference* option would allow any answer that matches the hard constraints (if exist) and any preferences (e.g., as in [11]) whereas an *at least N preferences* option would restrict candidate answers to those that satisfy at least N preferences (e.g., as in [14]).
- a *best-answer specification* that describes which of the candidate answers should be output. For instance, a *best-n* answer would comprise the best n tuples. If n=1, then only the tuples that are of the highest interest to the user would be returned.

The more the new query components are sealed from the details of their actual implementation, the easier is for users to understand and formulate queries with preferences. For instance, the *best* answer may vary and different algorithms may be required for answer computation depending on whether preferences are quantitatively or qualitatively captured. However, when formulating a preference-aware query, a user just needs to describe how much of the answer she is interested in seeing without caring about such details. This is also true, to some extent, for traditional queries: for example, a user does not care how a join is executed. Finally, a transparent preference-aware query model makes application development easier since the interface to a preference-aware database can be standardized even if the underlying implementation of the engine may vary.

4 Implementations of Preference-Aware Query Answering

In implementing preference-aware query processing, there are three possibilities: (*i*) query translation, (*ii*) special evaluation algorithms for implementing preference operators on-top of the DBMS, and (*iii*) native implementation, i.e., inside the database engine using specific physical operators and algorithms. We call the first two approaches plug-in since they are implemented on top of the database engine.

The straightforward approach to preference-aware query processing is to translate queries with preferences to conventional queries and execute them over the DBMS. Query translation conceptually involves the following steps (e.g., [4, 12, 14]): (*query rewriting*) the preferences are integrated as standard query conditions in the query producing a set of new queries, (*materialization*) the new queries are executed and (*aggregation*) the partial results are combined into a single ranked list. This is for example the approach taken in the implementation of Preference SQL, where preference queries are translated into standard SQL queries [12] as well as for integrating user preferences into queries for personalizing query results (e.g., [14]).

There is a large number of special evaluation algorithms for implementing preference operators (e.g., top-n [10] and winnow [2, 4]). For retrieving the top-n tuples of a relation, the basic idea has been captured in the algorithm called *FA* [5] and has been improved by several subsequent algorithms (e.g., [9]). These algorithms can be thought of as operating on the aggregation phase of query translation [16]. *FA* performs a sorted access to each partial list of results until there is a set of tuples, such that each of these tuples has been seen in each list. Then, for each tuple seen, *FA* performs random accesses to retrieve its missing scores and compute an aggregate score. Finally, the top-n tuples based on their aggregate scores are selected.

The naïve way to compute the winnow of a relation is to apply a basic nested-loop method that compares each tuple in the relation with every other tuple. This method works for every type of preference relation but requires scanning the whole relation for each tuple, which becomes inefficient for large datasets. Several improved algorithms have been proposed over this basic version (e.g., [2]) that employ techniques such as indexing (e.g., [21]) to avoid scanning the whole relation.

These special evaluation algorithms are typically implemented on top of the DBMS on the returned query results as standalone programs. In order to facilitate the integration of any preference operator (e.g., top-n, skyline) into the database engine with minimal effort, FlexPref is a system that allows registering preference methods through a set of functions that define rules for determining the most preferred tuples [15]. Once a preference method is injected into the database, it can be used for querying the database.

In the case of implementing preference operators inside the database engine, a set of algebraic rules that characterize the interaction of winnow with the standard relational operators, such as commutativity, are provided in [4] and can be used for query optimization. However, few actual native implementations exist. The rank operator [16] is an extension to relational algebra that enables pipelining and hence optimizing top-n queries. A hybrid approach to implementing Preference SQL that provides support for preference-aware query processing within the database engine has been also proposed [7].

Both native and plug-in approaches have pros and cons. Plug-in approaches rely on the existing Boolean query model, which is preference-agnostic. Their main advantage lies in the fact that they require no modification of the database code. They are easier to implement, and they leverage existing, invested, technology. On the downside, treating the DBMS as a black box means that there is no access to internal database operators. Consequently, these methods do not lend themselves to finer-grained dynamic optimization at the operator level based on the features of the query and the preferences. This has an impact on query performance and scalability.

In particular, in query translation, the execution plan for a query with preferences is composed outside the database engine. This external query plan is a high-level execution plan that describes the queries sent to the database, materializations of intermediate results and how partial results are combined in order to generate the answer to a query with preferences. Inevitably, possible query optimizations are more coarse-grained such as reducing the number of queries sent to the DBMS or the data processed outside the DBMS engine. Implementations of preference operators as programs outside the DBMS work as a post-processing step after partial query results have been generated, i.e., primarily improving the aggregation phase.

A fully native implementation of preference-aware query processing requires modifying the database query engine so that queries with preferences can be directly expressed in the query model supported by the database itself and query processing is entirely pushed inside the database engine. Native implementations of operators such as the rank operator [16] represent a significant step towards this direction. However, they mainly aim at

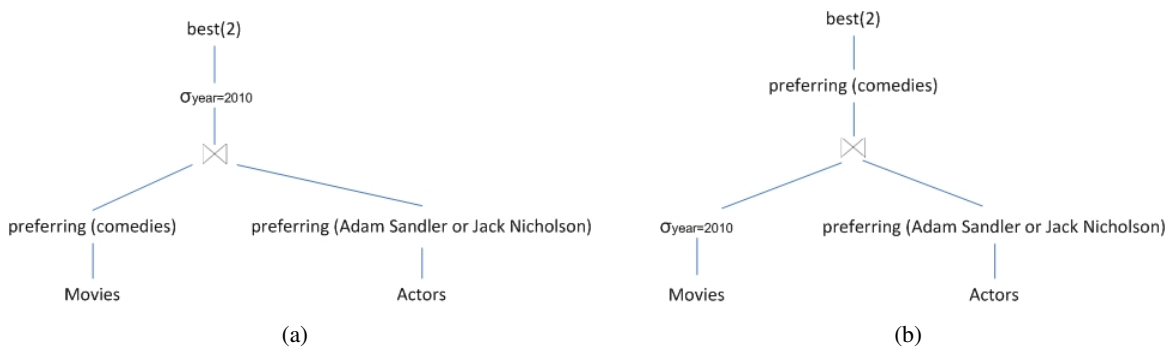

Figure 2: Preference-aware query plans

pushing aggregation inside the database engine while preference evaluation may be still performed externally.

To illustrate the benefits of a fully native implementation, let us assume a preference-aware DBMS. A hypothetical internal query plan (along the lines of the query model outlined in Section 3) is shown in Fig. 2(a). There are two preference-related operators combined with the classical relational operators. The 'preferring' operator applies a preference on a relation. For instance, if we have a quantitative preference that associates comedies with a score 0.8, the 'preferring' operator would assign this score to comedies. The 'best(n)' operator selects the n most preferred tuples. Such a query plan can be dynamically optimized taking into account the properties of each operator. For example, we could push the filter down the tree, apply the most selective preference first (that is the one referring to actors) or apply the 'best(n)' operator to each of the input relations to restrict the size of intermediate results. Fig. 2(b) illustrates some possibilities.

Another benefit of a native implementation is that query expressivity is not restricted by having a preference-agnostic database model hidden behind a query translation or special evaluation mechanism. Instead, queries with preferences are precisely defined by the preference-aware query model supported by the database. An important challenge is how expressivity and performance can be balanced. Depending on whether a qualitative or quantitative approach is followed, the new operators may have different properties enabling different optimizations. Making the operators very generic may limit the range of optimizations and may weigh down the benefits of a generic implementation. A careful design of the query optimizer is required.

5 Personalized Queries

In a preference-aware database, queries with preferences could be explicitly issued by users. For example, we can imagine a user interface that allows users to specify hard and soft criteria for movies they would like to see. Querying with preferences is a more relaxed search paradigm but it does not ensure that empty or too many answers will not be generated. To a great extent, it is still up to the user to formulate the right queries. In addition, user preferences may be ephemeral coming along with a particular request or long-term. For example, Bob is not a fan of R. Zelweger. This could be considered a long-term preference. A personalized preference-aware database system would store and maintain user preferences along with data. Furthermore, given a query and an ephemeral or long-term user profile, a personalized DBMS would dynamically decide how to best answer the query in order to satisfy the user information need. For instance, it could dynamically decide that for query Q_1 it does not make sense to show movies with actress R. Zelweger to Bob. Given the proliferation of applications that rely on user preferences to provide a personalized experience, there is an ongoing need for such personalized database systems that provide storage and processing of user preferences.

Dynamically modifying a query taking into account user preferences is termed *query personalization* [14]. Existing works implement personalized queries on top of the exact-match query model rather than a preference-aware one. Their most important limitation is that the effect of query personalization is manually determined

(e.g., by specifying the number of preferences that the answer should satisfy [14]) and they focus on how to retrieve these preferences from the profile and generate the expected personalized results.

However, query personalization needs to be considered from a fresh perspective as a complex dynamic optimization problem. There are several possible queries that can be generated and executed in place of the initial user query by applying query transformations such as: (*i*) some of the hard conditions of the query may be converted to soft conditions; (*ii*) preferences may be added in the query; (*iii*) preferences may replace some of the hard conditions. Which personalization option to follow, what preferences to consider related in the context of a particular query and what makes the best answer to a query for a particular user are shaped by several factors. The DBMS needs to consider the characteristics of the query and the possible effect of the preferences on the query. For example, if a query returns just a few results (like a query about movies with Meg Ryan), then user preferences may be applied only for result ranking. For a too broad query (e.g., a query on comedies), query personalization may use preferences to focus the result set. However, not all preferences may be applicable. For instance, preferences related to dramas may lead to empty answers if combined with a query about comedies. The query context is also important. For example, when a user is browsing movies, query personalization may be less aggressive than when the user explicitly asks for personalized suggestions.

Ideally, a personalized database system should help users find interesting answers that are natural given their queries. Answers that cannot be justified or explained in the context of their initial query or seem random are bound to confuse users. For example, if the user has asked for new comedies she has not watched, then if there are no such comedies, it may make sense to show zero results than suggesting watching a drama. Finally, some control may be given to the user at the application level in order to express the desired extent of personalization.

6 Conclusions

Querying with preferences provides a relaxed query paradigm compared to the exact-match database query model and can help generate answers tailored to each individual. There is substantial work in this area but no single preference-aware query model has been widely adopted in the database world. We believe that the key to making this happen is coming up with a preference-aware query model that is simple and as transparent to preference representation and implementation details as possible. Furthermore, in order to fully unleash the power of a preference-aware query model and exploit query optimization opportunities in the database core, such a model should be natively supported by the database engine. Pushing preferences inside the DBMS will also provide a transparent unified interface to applications and make application development lighter since much of the preference-related logic will be supported by the underlying database system.

Embedding preferences in database queries is a big step forward but alone it will not help users find the 'right' answers to their queries. A personalized database system would actually help by cleverly modifying queries considering user preferences, whenever available, to generate customized answers. However, from a traditional DBMS to a personalized preference-aware DBMS, there is a long way to go, and it goes beyond disguising the elephant to make it look like a chameleon.

References

[1] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD*, pages 297–306, 2000.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[3] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, pages 503–514, 2006.

[4] J. Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[6] P. C. Fishburn. Preference structures and their numerical representations. *Theoretical Computer Science*, 217(2):359–383, 1999.

[7] B. Hafenrichter and W. Kießling. Optimization of relational preference queries. In *ADC*, pages 175–184, 2005.

[8] S. O. Hansson. Preference logic. *Handbook of Philosophical Logic (D. Gabbay, Ed.)*, 8, 2001.

[9] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[10] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[11] W. Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

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

[13] G. Koutrika and Y. Ioannidis. Personalizing queries based on networks of composite preferences. *ACM Trans. Database Syst.*, 35(2), 2010.

[14] G. Koutrika and Y. E. Ioannidis. Personalized queries under a generalized preference model. In *ICDE*, pages 841–852, 2005.

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

[16] 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.

[17] J. Pitkow, H. Schtze, T. Cass, R. Cooley, D. Turnbull, A. Edmonds, E. Adar, and T. Breuel. Personalized search. *Comm. of the ACM*, 45(9), 2002.

[18] K. R. Scherer. What are emotions? and how can they be measured? *Social Science Information*, 44:695–729, 2005.

[19] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM Trans. Database Syst.*, 36(4), 2011.

[20] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Adding context to preferences. In *ICDE*, pages 846–855, 2007.

[21] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[22] M. P. Wellman and J. Doyle. Preferential semantics for goals. In *Proc. of AAAI*, pages 698–703, 1991.

[23] X. Zhang and J. Chomicki. Preference queries over sets. In *ICDE*, 2011.