The CareDB Context and Preference-Aware Database System

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

[1]Microsoft Research, Redmond, WA, USA, justin.levandoski@microsoft.com
[2]Alexandria University, Alexandria, Egypt, khalefa@cs.umn.edu
[3]University of Minnesota, Minneapolis, MN, USA, mokbel@cs.umn.edu

ABSTRACT

This paper provides an overview of the *CareDB* context and preference-aware database system. *CareDB* provides efficient and scalable *personalized* query answers to users based on their preferences and current surrounding context. Traditional relational database system employ rigid "all or nothing" semantics when answering queries. *CareDB* moves beyond such rigidness by providing support for "preference-aware" query processing methods. Specifically, *CareDB* supports a plethora of multi-objective preference methods capable of finding the "best alternatives" according to users' given preference objectives. This paper first describes the architecture of the *CareDB* system. It then describes the details for three of CareDB's novel query processing characteristics: (1) a generic and extensible preference-aware query processing engine, (2) a framework to gracefully handle contextual attributes that are expensive to retrieve, and (3) a framework to efficiently process queries over uncertain contextual data. Finally, it describes a prototype of the CareDB system and discusses interesting future research directions in personalized systems.

1. INTRODUCTION

Currently, database systems are extremely rigid: a user submits a query with a set of constraints to the database, and the system returns a set of answers that are exact matches for the constraints. In the worst case, the database may return no answers if the given constraints are too restrictive. For many application scenarios (e.g., location-based services, point-of-interest finders), users want from the database only a few "best" answers according to their personal preferences and context (e.g., location, weather). For instance, when searching for a restaurant, a user may want the database to return only five restaurants that present the best trade-off between minimizing the price and travel time to the restaurant, while maximizing the restaurant rating.

In the database literature, a number of multi-objective preference methods have been proposed that are capable of evaluating a set of user preference constraints. Examples of these methods include top-k [6], skylines [3], hybrid multi-object methods [2], k-dominance [4], k-frequency [5], and top-k dominance [22]. In general, the point of proposing new preference methods is to challenge the notion of "best" answers. Given the large number of preference methods already proposed, and likely to be created in the future, a fundamental research issue has become how to embed the semantics of each preference method within a database management system that may execute arbitrary queries composed of relational operators (e.g., selection, joins).

Another important consideration is how to integrate dynamic *contextual data* into preference query processing. Contextual data refers to any interesting data about the user or her environment that can help refine a preference answer. Such data is readily available through third-party web services. For instance, when searching for restaurants, a user may want to take into account travel time (e.g., using Bing or Google Maps) or restaurant reviews (e.g., Yelp). While useful, this contextual data poses two main problems when integrated with preference query processing: (1) the contextual attributes are *expensive* to derive relative to static data stored locally in a database (e.g., retrieved from a third party over a network) and (2) contextual data may contain uncertainty (e.g., restaurant prices reported as a range).

The CareDB project, started at the University of Minnesota in 2008, addresses the goal of embedding comprehensive support for context and preference-aware query processing within a database system. This paper describes the *CareDB* context and preference-aware database system that was built as a result of this project. *CareDB* addressees two *core* systems challenges: (1) support for various multi-objective preference evaluation methods (e.g., skyline, top-k) within the query processor and (2) integration of surrounding contextual data (e.g., traffic, weather) within the query processor, calling for support for gracefully handling expensive attributes and uncertain data. *CareDB* is a complete database system; all technical details discussed in this paper have been realized and experimentally evaluated in the PostgreSQL [19] open-source database system.

The rest of this paper describes how *CareDB* addresses the core systems research challenges behind embedding context and preference within a database query processor. Section 2 provides an overview of the *CareDB* architecture. Section 3 describes the novel technical features of *CareDB*. Section 4 describes a *CareDB* prototype. Finally, Section 5 discusses future research directions based on our experience building *CareDB*, while Section 6 concludes this paper.

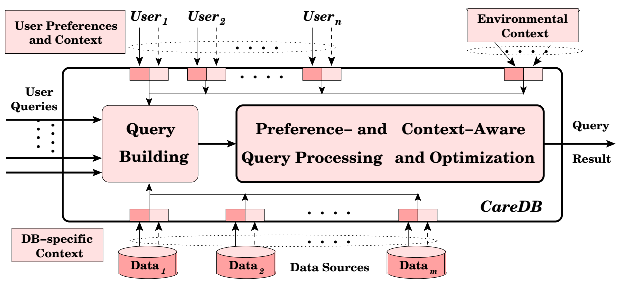
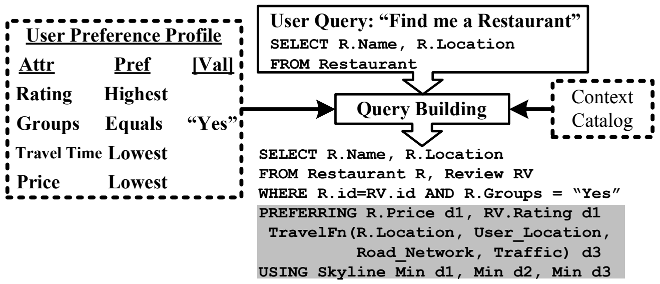


Figure 1: *CareDB* Architecture



User Preference Profile

Attr	Pref	[Val]
Rating	Highest	
Groups	Equals	"Yes"
Travel Time	Lowest	
Price	Lowest	

User Query: "Find me a Restaurant"

```
SELECT R.Name, R.Location
FROM Restaurant
```

```
SELECT R.Name, R.Location
FROM Restaurant R, Review RV
WHERE R.id=RV.id AND R.Groups = "Yes"
PREFERRING R.Price d1, RV.Rating d1
 TravelFn(R.Location, User_Location,
          Road_Network, Traffic) d3
USING Skyline Min d1, Min d2, Min d3
```

Figure 2: Preference queries in *CareDB*

2. CareDB ARCHITECTURE

This section provides an overview of the *CareDB* context and preference-aware database system, depicted in Figure 1.

Input. Besides queries expressed in SQL, *CareDB* takes *preference* and *contextual* data as input. Preferences are specified by a user and stored in a profile. In *CareDB*, a single user preference maps to single data attribute. The structure of a preference is `(Attribute, Preference, [Value])`, where *attribute* is a single data attribute, *preference* describes a user "wish" for that attribute, and *value* (numeric, categorical, or boolean) is optional determined by the type of preference. Preferences for a user are stored in their *preference profile*. The available preferences are based on the theoretical foundation of *PreferenceSQL* [12]. Preferences can be *hard* (e.g., equals) or *soft* (e.g., highest, lowest), and specified over numeric or categorical attributes. Furthermore, the user may specify a ranking function (either user-defined or built-in) over multiple attributes in order to perform top-k preference evaluation.

In addition, *CareDB* has three input context types, each context can be either *static* (rarely changed) or *dynamic* (frequently changing)

1. *User context.* User context is any extra information about a user. Examples of static user context data include income, profession, and age while dynamic attributes include current user location or status (e.g., "at home", "in meeting").

2. *Database context.* Database context refers to data sources (e.g., restaurant, hotel, and taxi databases) that are registered with *CareDB*, representing data in the domain a user wishes to query. As an example, for a restaurant database, static context data includes price, rating, and operating hours while dynamic context includes current waiting time.

3. *Environment context.* Environment context is any information about the user's surrounding environment, assumed to be stored at a third party and consulted by the query processor. A dynamic environmental context includes road traffic, while a relatively static context includes weather information.

Query Building Module and Preference Queries. The purpose of the query building module (rounded square in Figure 1) is to *personalize* queries for each user such that the *best* answers are returned. The user submits simple SQL queries without constraints (e.g., "Find me a restaurant"). The query building module creates *preference queries* by augmenting the submitted query with the preference constraints stored in the user's preference profile. *Preference queries* contain traditional relational constraints (e.g., selection conditions), as well as new preference constraints added to a preferring clause. In general, hard preference constraints are added to the where clause while soft preference constraints are specified in the preferring clause. Meanwhile, a using clause specifies what preference method will be used to evaluate the preference constraints to produce an answer. An example preference query for restaurants is given in Figure 2 that employs the skyline method in the using clause to produce a preference answer.

Query Processor. The preference and context-aware query processing and optimization engine is responsible for executing preference queries in *CareDB*. The main novelty of *CareDB* lies in this query processing engine. The main responsibilities of this module are as follows. (1) Embed various types of preference-aware query processing within a relational database engine. Specifically, we aim to support various types of multi-objective preference methods, e.g., skylines [3], k-dominance [4], top-k dominance [22]. Each method accepts preference objectives like those specified in the preferring clause of the query in Figure 2. However, given the same constraints, each method may produce a different preference answer for a given data set. (2) Support the integration of context-aware query processing. Contextual data (e.g., traffic and weather information) is assumed to be retrieved from a third-party source and expensive to derive relative to data stored locally in the database. (3) Support preference and context-aware query evaluation that involves *uncertain* data.

3. CareDB TECHNICAL FEATURES

A distinguishing feature of *CareDB* is its integration of preference and context concepts *inside* the database query processor. In this section, we describe the details of three novel systems features of *CareDB*: (1) *FlexPref*: the generic and extensible preference query processing engine of *CareDB* that includes an efficient preference-aware join operator, (2) a preference query processing framework for efficiently handling computationally-bound contextual data, and (3) a framework for efficiently answering preference queries for uncertain data.

3.1 FlexPref: An Extensible Preference Query Processing Framework

CareDB queries can be evaluated using any number of preference methods (e.g., skyline, top-k, k-dominance) based on the constraints given in the preferring clause. Thus, the

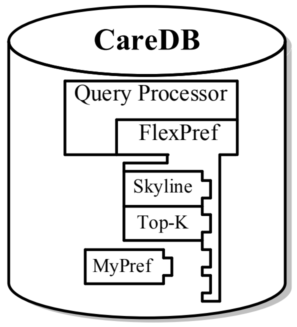


Figure 3: ***FlexPref***

query processor must be aware of how to evaluate any of these methods. One approach is to create a user-defined-function that evaluates preference *on-top* of a query plan. A second approach is to create *custom* operators for each preference method that can be integrated with the query processor. *CareDB* takes a third approach by implementing *FlexPref* [15], a *general* and *extensible* framework for implementing preference evaluation methods inside the query processor. Figure 3 relates the main idea of *FlexPref*. The framework is built into the PostgreSQL query processor, and *only FlexPref* touches the query processor. Each new preference method added to the system is "plugged into" *FlexPref* by registering only *three* functions: (1) PairwiseCompare: given two data objects P and Q, update the score of P and return whether P or Q can never be a preferred object based on a pairwise comparison of objects. (2) IsPreferredObject: given a data object P and a set of preferred objects S, return true if P is a preferred object and can be added to S, false otherwise. (3) AddPreferredToSet: given a data object P and a preference set S, add P to S and remove or rearrange objects from S, if necessary.

The main idea behind FlexPref is *separation of duties*. (1) The *registered functions*, specific to each preference method, define the semantics of the method's preference criteria. These functions define *how* one object is qualitatively better than the other. These functions are *not* aware of the internals of the query processor. (2) The *generalized* framework is responsible for efficient preference query processing by injecting preference evaluation as close to the native data operators as possible (i.e., selections, joins). The generalized framework uses the registered functions to evaluate the semantics of the specific preference method. With *FlexPref*, a preference evaluation method can "live" inside the query processor with minimal implementation effort compared to a *custom* approach. *FlexPref* consists of a set of generic, extensible relational operators. The basic idea is that each operator is written in terms of the extensible functions. The operator implements the common query processing functionality common across all supported multi-objective preference methods. During query runtime, the appropriate plugin functions for a specific preference method are used by the operators to evaluate the semantics of that method. In the rest of this section, we discuss three operators that make up the FlexPref framework.

3.1.1 Selection Operator

The selection operator produces the preference answer from data stored in a single table. The basic idea of the

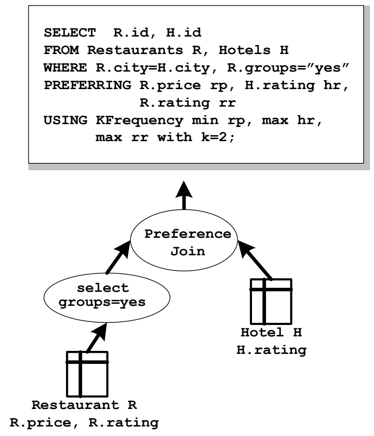

```
SELECT  R.id, H.id
FROM Restaurants R, Hotels H
WHERE R.city=H.city, R.groups="yes"
PREFERRING R.price rp, H.rating hr,
           R.rating rr
USING KFrequency min rp, max hr,
      max rr with k=2;
```

Figure 4: Preference join

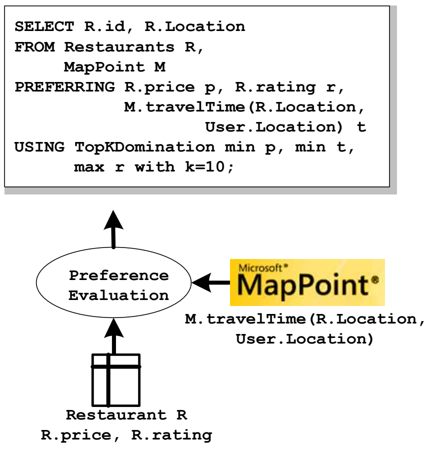
selection operator is to implement a block-nested loop algorithm to execute single-table preference evaluation (assuming data is not indexed). The operator compares tuples pairwise, using the plugin functions to evaluate the specific preference semantics, and incrementally builds a preference answer set.

3.1.2 Join Operator

FlexPref integrates preference evaluation with the join operation, resulting in an operator named PrefJoin [11, 15]. For example, the preference query depicted in Figure 4 involves a join between the Restaurant and Review relations, where preference evaluation is performed using attributes from both relations. A naive method to implement a preference-aware join would be join all necessary data and perform preference evaluation afterward. PrefJoin improves upon this naive strategy by using the extensible plugin functions to prune tuples *before* the join that have no chance of contributing to the final preference answer. The PrefJoin algorithm consists of four phases, namely, *local pruning*, *metadata preparation*, *join*, and *refinement*. The *local pruning* phase filters out, from each input relation those tuples that are guaranteed not to be in the final preference set. The *metadata preparation* phase associates metadata with each remaining tuple used to optimize the join. The *join* phase uses the metadata to decide on which tuples should be joined together. Finally, the *refinement* phase finds the final preference set from the output of the join phase. These steps enhance join performance as joins are usually non-reductive, thus pruning tuples early reduces the output of the join, which in turn means less data must be processed in subsequent operations after the join.

3.1.3 Sorted List Operator

If all attributes in a preference query are available in sorted order (e.g., data stored using the decomposed storage model [7]), *FlexPref* takes advantage of this property through a sorted list operator. The basic idea of this operator is to implement a general-case threshold algorithm, originally made popular for top-k ranking (e.g., Fagin's TA algorithm [8]). Tuples are read, one-by-one, in round-robin fashion from each list. Processing ends, producing a complete and correct preference answer, once a stopping condition is met. Different preference methods may have differ-



```
SELECT R.id, R.Location
FROM Restaurants R,
     MapPoint M
PREFERRING R.price p, R.rating r,
           M.travelTime(R.Location,
                        User.Location) t
USING TopKDomination min p, min t,
      max r with k=10;
```

Figure 5: Expensive attribute query

ent stopping conditions, thus the extensible plugin functions are used to determine when to stop round-robin processing. The advantage of the sorted list operator is that a complete and correct preference answer can be computed after reading only a portion of the sorted data, which reduces the I/O overhead compared to processing queries over unsorted, non-indexed data.

3.2 Query Processing with Expensive Contextual Data

In *CareDB*, it is assumed that some attributes will be expensive to derive, as determining their value may require extensive computations (e.g., road network travel time), or require retrieval from a third party (e.g., remote web service). Figure 5 gives an example query (and plan) to find a preferred restaurant using the top-k domination method [22], where attributes *price* and *rating* are stored in a local relation, while the *travel time* attribute is requested from the Microsoft MapPoint [17] web service based on the restaurant and user locations.

In this case, attributes retrieved through third-party data sources (e.g,. MapPoint) are considered *expensive* due to the network overhead/delay in transmitting the values of this attribute to the query processing source. Meanwhile, processing data stored locally at the query processor is considered a relatively cheap operation. Coupling preference query processing with a mix of "cheap" and "expensive" attributes changes the algorithmic cost model of preference query processing. As a concrete example, we ran a simple experiment in CareDB (implemented in PostgreSQL) comparing retrieval costs of location and third-party data. The retrieval of a single expensive attribute (driving time from the Microsoft MapPoint web service [17]) takes 502 ms. Alternatively, the time needed to read a cold and hot 8 KB buffer page from disk is 27 ms and 0.0047 ms, respectively. Clearly, local DBMS operations incur order of magnitudes less cost than retrieving a *single* expensive attribute. Under these circumstances, the preference-aware query processor should avoid computing the expensive attributes whenever possible.

The *CareDB* query processor is designed to take these challenges into account. *CareDB* employs a preference evaluation operator that computes the preference answer by retrieving as few expensive data attributes as possible [16]. The main idea is to employ a three-phase query processing framework outlined in Figure 6, consisting of an the *initialization* phase, *pruning* phase, and *cleaning* phase. Each phase has a *computation* step applied to the "cheap" attributes and a *request* step that issues either a random-access or range request to retrieve "expensive" attributes.

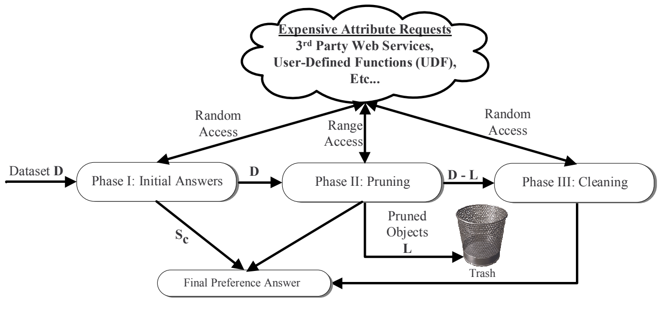


Figure 6: Solution overview

Phase I: *initialization.* Given a dataset D, Phase I forms an initial query answer (abbr. S_c) by running the preference query over the "cheap" (i.e., local) attributes. A random access request then retrieves the "expensive" attributes for objects in S_c. Phase I does not incur unnecessary requests, i.e., requests for expensive attributes associated with objects not in the preference answer, as it retrieves only the expensive attributes for those objects that are guaranteed to be in the final answer.

Phase II: *pruning.* Given the dataset D from Phase I, this phase performs three main operations: (a) Making a range request to retrieve the expensive attributes for a small sample of objects that are *not* in the initial answer S_c, (b) Creating a pruning set P by combining the returned objects from the range request with some of the objects in S_c. A set of objects $M \subseteq (P - S_c)$ are added to the final preference answer at this point. (c) Using P to prune a set of *incomplete* objects L that are guaranteed not to be in the final answer regardless of their expensive attribute values. Thus, the efficiency of our framework is to *maximize* the number of objects in L.

Phase III: *cleaning* This final phase takes as input the dataset $(D - L)$, and computes a final answer by first making a random request for remaining incomplete objects in $D - (L \cup S_c)$. Any dominated objects are then discarded. Any remaining objects are added to the final preference answer. Ideally, Phase III is unnecessary as all incomplete (and non-preferred) objects would be pruned by Phase II. Realistically, Phase III may incur some *unnecessary requests*.

3.3 Handling Data Uncertainty in CareDB

Given the growing number of applications that generate uncertain data (e.g., e.g., sensors, human entry errors), it is likely that some data registered with *CareDB* will contain uncertainty. Thus, *CareDB* employs a query processing framework, named *UPref* [10], capable of answering preference queries over data containing a mix of certain and uncertain attributes. *UPref* assumes uncertain attributes are represented as a continuous range of values, common in many real-life applications (e.g., biological data, spatial databases, sensor monitoring, and location-based services).

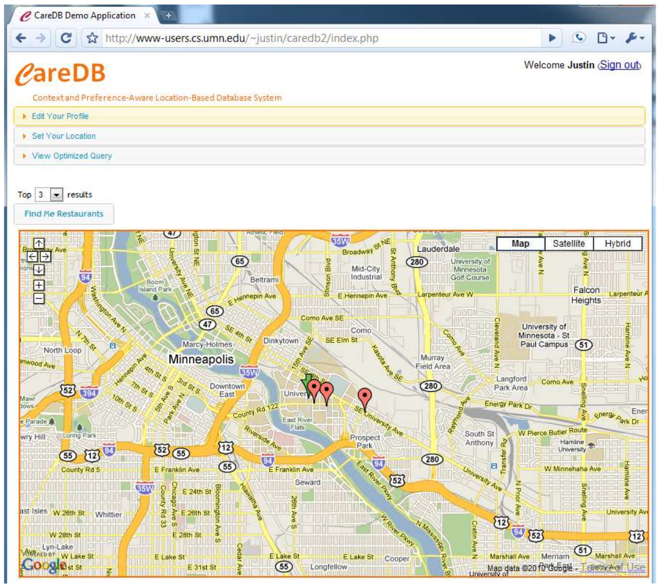

CareDB Demo Application
http://www-users.cs.umn.edu/~justin/caredb2/index.php
CareDB
Welcome Justin (Sign out)
Context and Preference-Aware Location-Based Database System
Edit Your Profile
Set Your Location
View Optimized Query
Top 3 results
Find Me Restaurants
Map
Satellite
Hybrid
Minneapolis
Downtown East
University of Minnesota - St Paul Campus
Prospect Park
Falcon Heights
Map data ©2011 Google - Terms of Use

Figure 7: *CareDB* Demo Application

UPref consists of two main phases for evaluating preference queries, following a *filter-refine* approach. The main reason behind using a *filter-refine* approach is that the brute-force approach involves the exponential process [13] of computing the probability that an object P in a data set D is a preferred object by comparing P to all other objects in D that affect P's probability. Thus, the first phase of *UPref* estimates an upper-bound probability for each object P to be a preferred object. If an object has an upper-bound probability that falls below the user-given threshold H, it is immediately filtered from the preference answer. Such an approach is advantageous as an object can be filtered *on-the-fly*, without comparing it with *every other* object.

The second phase of *UPref* computes a final preference probability for all objects returned from Phase I within a user-given tolerance. Phase II employs a novel and *efficient* probability calculation method that only performs as much computation *as is necessary* to guarantee a final probability calculation error falls within the user given tolerance. This efficiency comes through breaking the uncertainty range of an object into *segments*, and using these segments to compute an upper and lower bound probability for an object. These probability bounds are then *tightened* by iteratively partitioning segments (making the bound calculations more precise) until the calculation error falls within the user given threshold, or the upper bound probability for an object becomes lower than the user given threshold value. Phase II wisely chooses a segment to partition that is guaranteed to maximally tighten an object's upper and lower bound, minimizing the number of iterations. Thus, no computation is wasted in calculating an object's final preference probability. Much like *FlexPref* and the expensive attribute framework, *UPref* is designed to be generic and extensible, capable of supporting many well-known preference methods within a *single* framework.

4. CAREDB PROTOTYPE

To demonstrate the usefulness and functionality of *CareDB*, we implemented a location-based restaurant and hotel finding application using the Google Maps API, depicted in Figure 7, which interacts with the *CareDB* server implemented within PostgreSQL [19]. In the application, users can set their *CareDB* preference profile explicitly using a profile editor window. The editor allows the user to specify their preference objectives, *as well as* the preference method used to evaluate these objectives. Since the *CareDB* query processing framework (i.e., *FlexPref*) is generic and extensible, we provide a number of different preference methods to the user (e.g., skyline [3], top-k [6], top-k domination [22]). To process queries, the application forwards a simple query to *CareDB* (e.g., "find me a restaurant") where it is injected with preference and context constraints based on the users's preference profile. *CareDB* returns (1) the personalized preference SQL query that was run on the database, which can be displayed the application using a drop-down menu, and (2) the personalized query answers that are displayed on an embedded Google Maps interface.

5. FUTURE DIRECTIONS

Our experience building *CareDB* sparked interest in many novel directions within the area of personalized systems. We now outline four interesting research directions on the border of data management systems and personalization.

Database support for recommender systems. Experts from the recommender system community do not consider using a database to implement recommendation logic using relational operators. However, recent work on benchmarking recommender system architectures [14] reveals that integrating databases and recommender systems allows existing recommenders to become more scalable and robust. Currently, most recommender systems assume all necessary data fits in memory, and truncate important data structures if this assumption does not hold, in turn decreasing recommendation accuracy [21]. Furthermore, some high-quality recommendation techniques (e.g., user-based collaborative filtering [20]) have been abandoned altogether for large data due to efficiency problems. To address these drawbacks, an interesting open research direction would study how databases can support integration of existing recommendation techniques. Specifically, this direction would investigate coupling recommendation techniques with relational operators, new storage approaches for recommender data, and adapting current database components to efficiently handle dynamic properties found in recommendation applications (e.g., high update rates for recommender data structures).

Location-aware personalization. The use of mobile, location-based service applications is on the rise, and many of these applications require personalization (e.g., personalized restaurant/store finders, location-based advertising). Location-based environments present two primary challenges for personalization. (1) *Continuous query processing*. Location-based applications are by definition mobile, and require support for continuous queries since personalized answers change with user movement. In this vein, an interesting direction would be to adapt *CareDB* to continuous query environments. Specifically, this direction would investigate generic continuous preference-aware database operators coupled with an adaptive optimization scheme, where the operator pipeline for long-running preference queries may require re-optimization and re-ordering due to changes in the query environment. (2) *Novel location-based personalization methods*. This direction would address personalization techniques intended specifically for location-based services. Specifically, this area would involve investigation of location-based recommendation techniques that take into account both the location of a candidate recommendation items (e.g., restaurants), as well as the spatial locality of community opinions in the system.

Geo-social systems. The flourishing worldwide use of powerful mobile devices (e.g., smart phones), coupled with social networking applications, has brought about geo-social systems (e.g., Foursquare, Facebook Places), whereby user activities in the system contain locations as well as traditional attributes (e.g., friend updates, location-based check-ins, restaurant reviews). Such systems are coming into vogue, and pose interesting data management research challenges [18]. The general open research issues in this area would consider how best to achieve: efficient and scalable delivery methods for spatial news feeds customized to user access patterns, ranking and summarization techniques for location-based news items personalized for each system user, and the creation of novel real-time geo-social analysis techniques. Each of these issues poses interesting scalability and efficiency challenges for systems that must scale to a large amount of concurrent users.

Real-time recommender systems. The advent of dynamic online applications (e.g., online news, social networks) challenges many of the traditional assumptions made by existing recommendation techniques. In these environments, there is a rapid turnover of candidate recommendation items (e.g., news stories, friend status updates in social networks), as well as a high rate of community feedback (e.g., news story "Diggs", Facebook "likes", Foursquare "check ins"). However, many of the well-known personalization and recommendation techniques, such as collaborative filtering [1, 9, 20, 21], are not built to handle such dynamic environments, as they employ a compute-intensive offline phase that builds a statistical model, incorporating new items and user feedback, in order to produce recommendations. Thus, an interesting area of research would be to study methods to adapt existing, high-quality recommendation techniques to cope with these new, dynamic environments.

6. CONCLUSION

This paper described the *CareDB* system, a full-fledged relational database system capable of providing *personalized* answers to users based on their preference and surrounding context. The architecture of *CareDB* was first presented that provided an overview of the input (user preferences and contextual information) along with the query processing objectives of *CareDB*. The paper then presented an overview of three novel technical features of *CareDB*, namely, an extensible query processing framework, the ability to gracefully handle contextual data that is expensive to derive at query runtime, and the ability to efficiently provide preference query processing support for uncertain data. An overview of the *CareDB* prototype was also given, along with a discussion of interesting future directions in personalized data management systems sparked by our experience in building *CareDB*.

7. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *TKDE*, 17(6):734–749, 2005.

[2] W.-T. Balke and U. Güntzer. Multi-objective Query Processing for Database Systems. In *VLDB*, 2004.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, 2001.

[4] C.-Y. Chan et al. Finding k-Dominant Skylines in High Dimensional Space. In *SIGMOD*, 2006.

[5] C.-Y. Chan et al. On High Dimensional Skylines. In *EDBT*, 2006.

[6] S. Chaudhuri and L. Gravano. Evaluating Top-K Selection Queries. In *VLDB*, 1999.

[7] G. P. Copeland and S. N. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[8] R. Fagin et al. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[9] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating Collaborative Filtering Recommender Systems. *ACM Transactions on Information Systems, TOIS*, 22(1):5–53, 2004.

[10] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Skyline Query Processing for Uncertain Data. In *CIKM*, 2010.

[11] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. PrefJoin: An Efficient Preference-Aware Join Operator. In *ICDE*, 2011.

[12] W. Kießling. Foundations of Preferences in Database Systems. In *VLDB*, 2002.

[13] C. Koch and D. Olteanu. Conditioning Probabilistic Databases. In *VLDB*, 2008.

[14] J. J. Levandoski, M. D. Ekstrand, M. J. Ludwig, A. Eldawy, M. F. Mokbel, and J. T. Riedl. RecBench: Benchmarks for Evaluating Performance of Recommender System Architectures. In *VLDB*, 2011.

[15] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. FlexPref: A Framework for Extensible Preference Evaluation in Database Systems. In *ICDE*, 2010.

[16] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. Preference Query Evaluation over Expensive Attributes. In *CIKM*, 2010.

[17] Microsoft MapPoint: http://www.microsoft.com/mappoint/.

[18] M. F. Mokbel et al. Personalization, Socialization, and Recommendations in Location-based Services 2.0. In *PersDB*, 2011.

[19] PostgreSQL: http://www.postgresql.org.

[20] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSWC*, 1994.

[21] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-Based Collaborative Filtering Recommendation Algorithms. In *Proceedings of the International World Wide Web Conference, WWW*, 2001.

[22] M. L. Yiu and N. Mamoulis. Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data. In *VLDB*, 2007.