An Overview of the CareDB Context and Preference-Aware Database System

Justin J. Levandoski   Mohamed E. Khalefa   Mohamed F. Mokbel

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN
{justin,khalefa,mokbel}@cs.umn.edu

Abstract

This article 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.* CareDB *moves beyond the* rigid *query processing semantics of traditional relational database systems, which employ a boolean "all or nothing" query model, and addresses 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 article describes the architecture of the* CareDB *system and describes the details for three of its 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.*

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 [5], skylines [2], hybrid multi-object methods [1], k-dominance [3], k-frequency [4], and top-k dominance [13]. 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).

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

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. Currently, 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 the Bing Maps web service) or restaurant reviews (e.g., using the Yelp web service). 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).

Toward the goal of embedding support for context and preference within a DBMS, this article describes *CareDB*: a context and preference-aware database system. *CareDB* addressees two fundamental *core* systems challenges: (1) support for various multi-objective preference evaluation methods (e.g., skyline, top-k, k-dominance) within the query processor and (2) integration of surrounding contextual data (e.g., current traffic, weather) within the query processing, calling for support for gracefully handling expensive attributes and uncertain data. *CareDB* is a complete database system, meaning all ideas discussed in this article have been realized and experimentally evaluated in the PostgreSQL [12] open-source database system.

In the rest of this article, we describe how *CareDB* addresses the core systems research challenges behind embedding context and preference within the database query processor. Section 2 provides an overview of the *CareDB* architecture. Section 3 describes the novel technical features of *CareDB*. Finally, Section 4 describes a *CareDB* prototype.

2 CareDB Overview

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

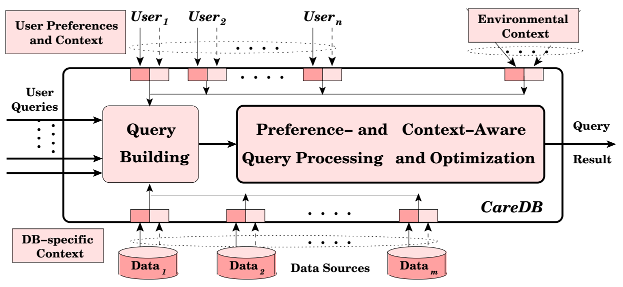
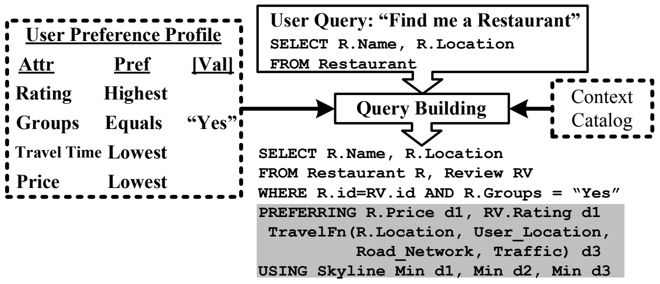


Figure 1: *CareDB* Architecture

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* [8]. 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.



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*

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 [2], k-dominance [3], top-k dominance [13]. 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 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 a *custom* implementation for each preference method that can be integrated with query operators. *CareDB* takes a third approach by implementing *FlexPref* [9], 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.

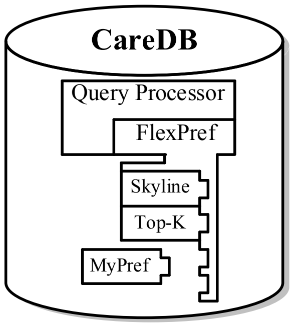
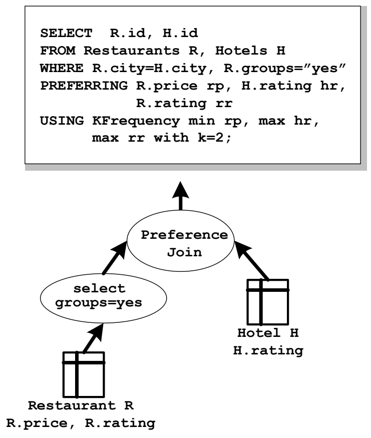


Figure 3: *FlexPref*

The main idea behind FlexPref is *separation of duties*. (1) The *registered functions*, specific to each preference method, define the semantics of the 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. Currently, *FlexPref* consists of the following three operators.

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



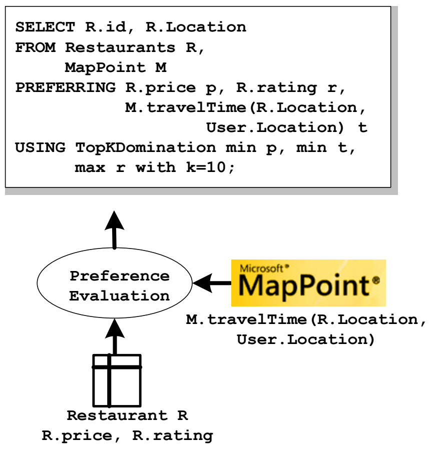
```
SELECT  R.id, H.id
FROM Restaurants R, Hotels H
WHERE R.city=H.city, R.groups="yes"
PREFERRING R.price rp, H.rating hr,
           R.rating rr
USING KFrequency min rp, max hr,
      max rr with k=2;
```

Figure 4: Preference join

Join. *FlexPref* integrates preference evaluation with the join operation, resulting in an operator named PrefJoin [7, 9]. 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.

3.2 Query Processing with Expensive Contextual Data

In *CareDB*, it is assumed that some attribute values will be expensive to derive, as the derived value may require extensive computations (e.g., road network travel time), or must be retrieved 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 [13], where attributes *price* and *rating* are stored in a local relation, while the *travel time* attribute is requested from the Microsoft MapPoint [11] web service based on the restaurant and user locations. Under these circumstances, computational overhead is dominated by deriving the expensive *travel time* attribute, thus the preference query processing operator should avoid computing these expensive attributes whenever possible.



```
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

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 [10]. The main idea is to first perform preference evaluation over *local* data attributes, forming a local answer set LA using "plug-in" functions to determine the semantics of the specific preference method used to execute the query (e.g., top-k domination). The operator then selectively requests expensive attributes for objects in LA guaranteed to be preference answers, and *prunes* objects that are guaranteed not to be preference answers. *CareDB* then attempts to make the minimum number of expensive attribute requests necessary to completely and correctly answer the preference query.

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* [6], 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). *UPref* associates a probability P with each object O, that represents the chance that P is an answer to the preference query.

To process queries, *UPref* assumes two system parameters: (1) A *tolerance* value Δ that specifies the maximum error allowed in calculating probability P, and (2) a *Threshold* value H, that each object probability must exceed in order to be a preference answer. *UPref* employs a two-phase filter-refine approach to processing preference queries over uncertain data. Phase I calculates an estimated upper-bound preference probability for each object, and *filters* objects on-the-fly that have an upper-bound probability that falls below the threshold H. Phase II computes a final preference probability for each candidate answer within a user-given tolerance Δ. Phase II employs a novel, efficient probability calculation method that performs only as much computation *as is needed* to guarantee the final preference probability for an object falls within Δ. 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.

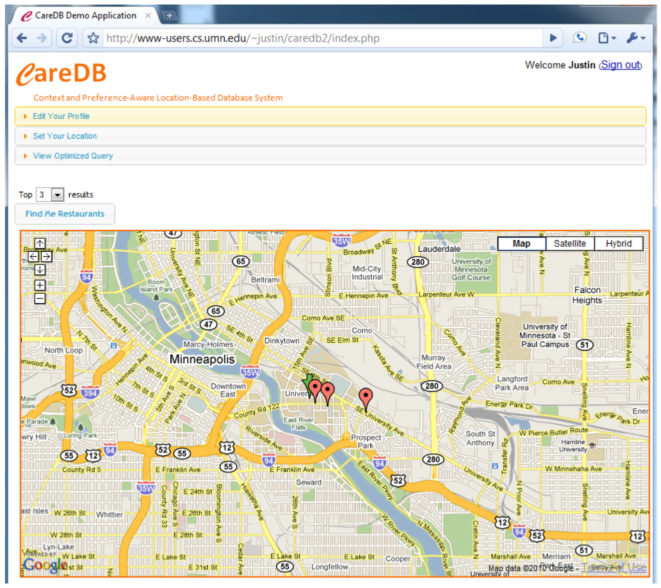


Figure 6: *CareDB* Demo Application

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 6, which interacts with the *CareDB* server implemented within PostgreSQL [12]. 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 [2], top-k [5], top-k domination [13]). 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.

References

[1] W.-T. Balke and U. Güntzer. Multi-objective Query Processing for Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2004.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2001.

[3] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang. Finding k-Dominant Skylines in High Dimensional Space. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2006.

[4] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang. On High Dimensional Skylines. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2006.

[5] S. Chaudhuri and L. Gravano. Evaluating Top-K Selection Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 1999.

[6] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Skyline Query Processing for Uncertain Data. In *Proceedings of the International Conference on Information and Knowledge Managemen, CIKM*, 2010.

[7] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. PrefJoin: An Efficient Preference-Aware Join Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2011.

[8] W. Kießling. Foundations of Preferences in Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2002.

[9] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. FlexPref: A Framework for Extensible Preference Evaluation in Database Systems. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2010.

[10] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. Preference Query Evaluation over Expensive Attributes. In *Proceedings of the International Conference on Information and Knowledge Managemen, CIKM*, 2010.

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

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

[13] M. L. Yiu and N. Mamoulis. Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2007.