CareDB: A Context and Preference-Aware Location-Based Database System

Justin J. Levandoski  Mohamed F. Mokbel  Mohamed E. Khalefa
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN
{justin,mokbel,khalefa@cs.umn.edu}

ABSTRACT
We demonstrate CareDB, a context and preference-aware database system. CareDB provides scalable personalized location-based services to users based on their preferences and current surrounding context. Unlike existing location-based database systems that answer queries based solely on proximity in distance, CareDB considers user preferences and various types of context in determining the answer to location-based queries. To this end, CareDB does not aim to define new location-based queries, instead, it aims to redefine the answer of existing location-based queries. To achieve its goals, CareDB has several distinguishing characteristics that revolve around a generic and extensible preference and context-aware query processing framework that addresses (a) scalable, efficient preference joins, (b) gracefully handling contextual attributes that are expensive to derive, and (c) support for uncertain attributes.

1. INTRODUCTION
Location-based services are viewed as the convergence of mobile device technologies, GIS/spatial databases, and the Internet. Location-based services aim to provide new services to their users based on the knowledge of their locations. Examples of these services include live traffic reports (“Let me know if there is congestion within five minutes of my route”) and store finders (“Where is my nearest restaurant”). A recent report from ABI Research indicated that the number of location-based services subscribers will be 315 Million by 2011 [1]. The flood of information generated by location-detection devices, along with the large number of mobile users of location-based services, calls for the integration of location-based service functionality with database systems.

Unfortunately, the system semantics of location-based databases are rigid as concepts of user “preference” and “context” are ignored. For example, when a user looks for a restaurant, she actually wants to find the “best” restaurant according to her current preferences and context. Existing location-based query processors reduce the meaning of “best” to be the “closest” in terms of pre-computed distances. If desired, preferences and/or context parameters are applied as afterthought queries over the returned result from location-based queries. The rigidity of current location-based query processors can be shown with a simple example where a user asks for five restaurants. After retrieving the answer (the nearest five restaurants), the user discovers that the first restaurant has an undesirably long wait, while the second restaurant does not match the user’s dietary restrictions. The third restaurant is outside of the user’s budget, while the fourth restaurant is closed. Finally, the route to the fifth restaurant is infeasible due to a traffic accident. Location-based services should be useful, and a more useful set of answers could have been given in the previous example had the database considered user preferences (e.g., dietary restrictions, budget) and contexts (e.g., time of day, traffic, waiting times).

It is our goal in this demo to enable the practical realization of location-based services that embed various forms of preferences and context into the core processing of location-based queries. To this end, we are not aiming to define new location-based queries, instead, we aim to redefine the answer of existing traditional location-based queries by incorporating various types of preferences and context. Due to resource limitations on mobile devices (e.g., small screen and limited processing), and the fact that users may be in unstable situations (e.g., driving), it is of essence to enhance the quality of the answer and limit the answer to only useful tuples according to the users’ preferences and context.

Toward the goal of adding preference and context to location-based services, i.e., enhancing the quality of answer by limiting the query result to only useful tuples, we propose the CareDB system: a context and preference-aware location-based database system. CareDB is a complete database system, implemented in PostgreSQL, that addresses the following research challenges:

1. Defining a taxonomy of preference and context types.
2. Generically supporting various preference evaluation methods at all levels of the query processor, including core query operators (e.g., join).
3. Integrating surrounding contextual data (e.g., current traffic, weather) in core preference query processing. Contextual data calls for retrieving some attributes from computationally-intense sources (e.g., third-parties).
4. Support for data uncertainty that is inherent in location-based applications (e.g., prices reported as a range, travel-time estimation reported with a tolerance error).

In the rest of this paper, we describe how CareDB addresses these research challenges as well as our demonstration scenario. Section 2 provides an overview of the CareDB architecture. Section 3 describes the novel technical features of CareDB. Finally, Section 4 covers our demonstration scenario that showcases CareDB in a real-life application scenario.
2. CareDB OVERVIEW

Our prototype of CareDB is implemented inside the PostgreSQL [10] database engine. Figure 1 gives an overview of the CareDB architecture.

CareDB Input. Besides queries, CareDB takes preferences and contextual data as input. Preferences are a set of users' tastes for data attributes in a particular domain. For example, whenever a user searches for a restaurant, her profile may store preferences for minimizing travel time and price, maximizing rating, and any constraints on dietary needs. User preferences are stored in a preference profile, and can be given to CareDB explicitly, or learned through the users history [12]. CareDB has three input context types: user context, database-specific context, and environmental context. Each context can be either static (rarely changed) or dynamic (frequently changing). Static/dynamic context is depicted by solid/dotted lines and dark/light gray rectangles, respectively, in Figure 1. User context is any extra information about a user. Static user context data can include income, profession, and age while dynamic attributes include current user location or status (e.g., "at home", "in meeting"). Database context refers to application-specific data sources (e.g., restaurant, hotel, and taxi databases) that are registered with CareDB. As an example, for a restaurant database, static context data includes price, rating, and operating hours while dynamic context includes current waiting time. Environmental context is any information about surrounding environment. This data is assumed to be stored at a third party and accessed by the query processor during query runtime by calling the data source through a remote API (e.g., web service interface). A dynamic environmental context includes traffic and travel time (e.g., from Yahoo Traffic [13]), while a relatively static context includes weather information (e.g., from NWS [9]).

Query building. 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 queries without constraints (e.g., “Find me a restaurant”). The query building module creates preference queries by augmenting the submitted query with the preferences stored in the user’s preference profile. We describe preference queries next.

Query Processor The novelty of CareDB lies within the preference and context-aware query processing module. The query processor takes as input preference queries. In previous work, we built FlexPref [7] that: (1) extended SQL syntax with a Preferring clause for specifying preference objectives (e.g., minimize price, maximize distance), and a Using clause that specifies which preference method should evaluate the objectives (e.g., skyline [2], top-k dominance [14], k-dominance [3]). Examples of this syntax are given in Figure 2. (2) Built a framework of generic, extensible operators inside the DBMS query processor to execute the preference query. In this demonstration, we focus on more novel features of CareDB that address query processing challenges beyond those covered by FlexPref, namely, providing a generic, extensible platform for preference evaluation with (a) efficient join operations,

![Figure 1: CareDB Architecture](image)

![Figure 2: Preference and context-aware query examples](image)

3. CareDB TECHNICAL FEATURES

Unlike previous work that addresses query processing challenges for specific preference methods (e.g., progressive skyline joins [6]), all CareDB features discussed in this section are built in a generic and extensible manner, capable of supporting multiple preference methods. The basic idea is to create a single, generic, operator for each feature (e.g., uncertainty, join), that performs computations common across preference methods. Each operator is extensible through “plug-in” functions that tell the operator about the semantics of a specific preference method [1]. From a systems perspective, this is a powerful method for implementing preference query processing in a database, as the engine need only be changed once, while existing and future preference methods can “plug-into” the existing framework.

3.1 Query Processing with Expensive Attributes

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 2(a) gives an example query (and plan) to find a preferred restaurant using the top-k dominance method [14], where attributes price and rating are stored in a local relation, while the travel time attribute is requested from the Microsoft MapPoint [8] 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.

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. 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 dominance). 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.

1We refer to [7] for details of the concept of extensible preference query processing in a DBMS engine, and initial work on simple query operations.
toring, and location-based services). The applications (e.g., biological data, spatial databases, sensor monitoring, and location-based services) of uncertain data, common in many real-life applications, can make it difficult to process preference queries efficiently. CareDB associates a probability $P$ with each object $O$, that gives the chance that $P$ is an answer to the preference query. CareDB uncertainty processing assumes two system parameters: (1) A tolerance value $\Delta$ 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. The CareDB uncertainty framework 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 $\Delta$. 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 $\Delta$.

4. DEMONSTRATION SCENARIO

We now outline the demonstration scenario, focusing on the application, data, queries, and walk-through scenarios for CareDB.

4.1 Application

The application we use is RestPref, a location-based restaurant and hotel finder application built specifically for this demonstration. RestPref, depicted in Figure 3, comes in two versions: (1) web-based, displayed in a standard browser or (2) mobile-based, as an iPhone application, both of which are on display in the demonstration. Both versions use CareDB as the backend database, which performs the query processing tasks. In RestPref, the user issues a simple query by pressing a button, we provide three buttons: (a) “Find me a restaurant”, (b) “Find me a hotel”, and (c) “Find me a hotel/restaurant pair”. The application forwards the simple query to CareDB where it is injected with preference and context constraints based on the user’s preference profile (covered shortly). CareDB returns (1) the personalized SQL query that was run on CareDB, which can be displayed in RecPref using a drop-down screen as displayed in Figure 4, and (2) the personalized query answers that are displayed on an embedded Google Maps interface.

4.2 Data, Preference Profiles, and Queries

Data. The data we use in RestPref is a set of restaurant and hotel data for Singapore. The restaurant data is stored in CareDB as relation Restaurant(id, name, price, rating, travel time). The travel time attribute is derived at query runtime using a remote call to Microsoft’s MapPoint web service [8] using the RestPref user’s current location, and is considered an expensive attribute. The hotel data is stored in CareDB as relation Hotel(id, name, rating, price); the price attribute for hotels is reported as a range (e.g., 100-200 dollars), and thus contains uncertainty.

CareDB then makes a minimum number of expensive attribute requests necessary to completely and correctly execute the preference query. In this demo, we provide sample queries that require requests to third-party web-service data in order to determine route travel time.

3.2 Generic Preference Join

Like most query processing scenarios for real-life applications, it is likely most data stored locally in the CareDB will not reside in a single table. For example, Figure 2(b) gives an example preference query (and plan) using the $k$-dominance preference method [3] for a user requesting a hotel and restaurant pair. The preference objectives are to minimize the restaurant price, while maximizing both restaurant and hotel ratings. The hotel and restaurant data is stored in separate tables, necessitating a join to answer the preference query. The naive method to answer this query is to perform the join, then perform preference evaluation. Current state-of-the-art join methods address a specific preference method (e.g., skyline join [6]), often assuming a specific index for progressive result generation [6, 11].

CareDB, on the other hand, employs PrefJoin, an efficient preference join operation that is generic for a wide variety of preference functions and does not assume the existence of any index structure. The goal of PrefJoin is to make the join operation aware of the required preference functionality through the “plug-in” functions, and hence the join operation would be able to early prune those tuples that have no chance of being a preferred object without actually doing the join operation. The PrefJoin algorithm consists of four phases, namely, Local Pruning, Data Preparation, Joining, and Refining. The Local Pruning phase filters out, from each input relation those tuples that are guaranteed not to be in the final preference set. The Data Preparation phase associates meta data with each non-filtered tuple that will be used to optimize the execution of the next phase. The Joining phase uses that meta data, computed in the previous phase, to decide on which tuples should be joined together. Finally, the Refining phase finds the final preference set from the output of the joining phase.

3.3 Query Processing with Data Uncertainty

It is likely that real-world data will contain uncertainty, thus CareDB is built to handle uncertain data. CareDB assumes uncertainty as a continuous range of values, common in many real-life applications (e.g., biological data, spatial databases, sensor monitoring, and location-based services). The uncertainty framework of
Preference Profile. In this demo, users can set their CareDB preference profile explicitly from RestPref using the profile editor window, as depicted in Figure 5. The editor allows the user to specify their preference objectives, as well as the preference method used to evaluate these objectives. Since our CareDB query processing framework is generic and extensible, we provide a number of different preference methods in this demo, including: skyline [2], top-k [5], top-k domination [14], k-dominance [3], and k-frequency [4].

Queries. As mentioned previously in Section 2, objectives listed in the preference profile determine the preference query executed by CareDB. For example, if the user profile contains objectives to maximize restaurant rating and minimize restaurant travel time for restaurants using the top-k domination method [14], and they push the “Find me a restaurant” button, the preference query executed by CareDB will be exactly the same as that given in Figure 2(a), requiring CareDB to process the travel time expensive attribute efficiently. Meanwhile, if a user profile contains objectives to minimize restaurant price and maximize rating, while maximizing hotel rating using the k-dominance method [3], and they press “Find me a restaurant/hotel pair” button, the query executed by CareDB will be exactly the same as that given in Figure 2(b), requiring CareDB to execute its efficient preference-aware join. Similarly, the uncertain preference query processing framework is invoked if the profile contains preferences for the hotel price attribute.

4.3 Walkthrough

The audience of the demo will be able to perform the following actions using CareDB and the RestPref application.

Canned scenarios. We provide three pre-set user preference profiles that cause CareDB to execute a preference query with (1) a preference-aware join between the hotel and restaurant tables, (2) an expensive attribute (travel time), and (3) uncertainty (hotel price). Each of these queries showcases one of the frameworks described in Sections 3.1 through 3.3. Each of the queries will be executed five times, each using a different preference function (e.g., skyline [2], top-k domination [14]). For each different preference function CareDB uses the same generic operator to process the query, while only the “plug-in” functions change according to the specific preference function.

Edit profile. The audience will be able to edit the preference profile to cause CareDB to execute an array of preference queries. For instance, the audience may choose to create a profile with objectives to minimize restaurant price and maximize rating, while minimizing hotel price and maximizing hotel rating, and press “find me a hotel/restaurant pair”, causing CareDB to combine its uncertainty and join functionality.

Backend access. Backend access to the CareDB server (implemented in PostgreSQL) is also available through a GUI client, as depicted in Figure 6. If interested, users can issue ad-hoc queries to CareDB through the backend. Users can also explore query plans - depicted visually in the GUI application - used to execute preference and context-aware queries. For example, Figure 6 depicts the query plan using CareDB preference join operator on a join-query between the restaurant and hotel relations.

Performance showcase. Our demo will showcase the performance benefits of the expensive attribute, join, and uncertainty query processing frameworks of CareDB. We will first disable the join and expensive attribute frameworks, forcing CareDB to execute the respective preference query in a naive manner. A naive join query will first perform the complete join, then evaluate the preference objectives. A naive expensive attribute query will first request all expensive attributes (i.e., travel time) from the third-party source, then evaluate the preference objectives. We will then enable the join and expensive attribute frameworks and execute the same queries. The audience will be able to view the performance benefit by the query processing times reported by the GUI backend client (Figure 6). To exhibit the performance benefits of the CareDB uncertainty framework, we implement a naive query processing operation that calculates the exact probability of each object (e.g., hotel) to be a preference answer. This naive method is compared to the CareDB filter-refine uncertainty query processing framework.

5. REFERENCES