Toward Context and Preference-Aware Location-based Services*

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

ABSTRACT

The explosive growth of location-detection devices, wireless communications, and mobile databases has resulted in the realization of *location-based services* as commercial products and research prototypes. Unfortunately, current location-based applications (e.g., store finders) are *rigid* as they are completely isolated from various concepts of user "preferences" and/or "context". Such rigidness results in non-suitable services (e.g., a vegetarian user may get a restaurant with non-vegetarian menu). In this paper, we introduce the system architecture of a Context and Preference-Aware Location-based Database Server (*CareDB*, for short), currently under development at University of Minnesota, that delivers *personalized* services to its customers based on the surrounding context. *CareDB* goes beyond the traditional scheme of "one size fits all" of existing location-aware database systems. Instead, *CareDB* tailors its functionalities and services based on the preference and context of each customer. Examples of services provided by *CareDB* include a restaurant finder application in which *CareDB* does not base its choice of restaurants solely on the user location. Instead, *CareDB* will base its choice on both the user location and surrounding context (e.g., user dietary restriction, user preferences, and road traffic conditions). Within the framework of *CareDB*, we discuss research challenges and directions towards an efficient and practical realization of context-aware location-based query processing. Namely, we discuss the challenges for designing user profiles, multi-objective query processing, context-aware query optimizers, context-aware query operators, and continuous queries.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

General Terms

Design, Architecture

*This work is supported in part by the National Science Foundation under Grants IIS 0811998, IIS0811935, and CNS0708604

1. INTRODUCTION

Combining the functionally of *map software* (e.g., Google maps, Microsoft MapPoint, Yahoo maps), *location-detection devices* (e.g., GPS antenna, cellular phones, and RFIDs), *personal handheld devices* (e.g., PDA and GPS), *wireless communication*, and *database systems* has resulted in the realization of *location-based services* as commercial products (e.g., see [27, 35]) and research prototypes (e.g., see [16, 30, 38]). The main promise of location-based services is to provide new services to their customers based on the knowledge of their locations. Examples of these services include continuous live traffic reports (*"Continuously, let me know if there is congestion within five minutes of my route"*), food and drink finder (*"Where is my nearest fast food restaurant"*), and location-based advertising (*"Send e-coupons to all cars that are within two miles of my gas station"*). According to the Cellular Telecommunication and Internet Association, CTIA, there are more than 240 Million wireless customers in the Unites States [10] while a recent research report from ABI Research indicated that the number of location-based services subscribers will be 315 Million by 2011 [1]. Due to the abundance of location-based data, location-based services have begun to integrate their functionality with database systems [19, 20, 26, 29]

Figure 1 gives a high level overview of location-based services where users issue various location-based queries (e.g., *"Where is my nearest restaurant"*) through their personal devices. Mobile devices are connected to a *database server* to submit their queries along with their locations that are retrieved through location-detection devices. Finally, the database server evaluates the user query based on the user location, the underlying map, and the locations of objects of interest (e.g., restaurants). At the user level, the users see the answer of their queries on the handheld devices guided by the map layout.

Unfortunately, the current state-of-the-art processing of location-based queries is **rigid**, as query processing completely isolates various forms of user "preferences" and/or "context". For example, in a restaurant finder application, the users actually want to find the *"best"* restaurant according to their current preferences and context. Existing location-based query processors reduce the meaning of *"best"* to be only the *"closest"* restaurant. Any query processing that produces results based on preference and or context is applied *after* the location-based database operations. In other words, preference and context are considered afterthought problems in terms of query processing. To show the rigidness of current techniques, consider the case

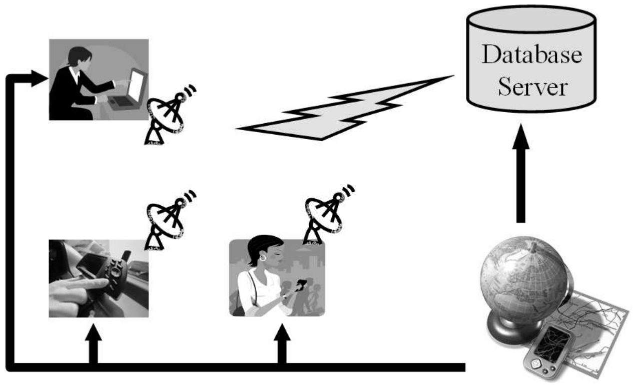


Figure 1: Location-based services

of a user asking about the nearest five restaurants. After retrieving the answer and going to the first restaurant, the user discovers that the restaurant has an unbearably long wait, while the second restaurant ends does not match the user dietary restrictions. Meanwhile, the third restaurant is too expensive for the user's budget, while the fourth restaurant is currently closed. Finally, the route to the fifth restaurant is infeasible as there is congestion due to a traffic accident. The rigidness of such an approach is due to two main reasons: (1) The lack of personalized customer services. For example, if two persons asking the same query in the same location, they will get the same answer, even if their personal preferences are different. (2) The lack of context awareness as the only considered context is the user location, while other kinds of context (e.g., weather, personal preference, and current road conditions) are completely ignored.

In this paper, we aim to raise the challenges and provide research directions to enable practical realization of preference and context-aware location-based services. The main idea is to embed various forms of preferences and context in 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 location-based queries.* As the query answer may be returned to the users on their mobile devices with limited screen capabilities, it is of essence to *enhance the quality of the answer* and limit the answer to only those tuples that are of major interest to the users according to their preferences and context. For example, we aim to force traditional restaurant finder queries to consider user preferences (e.g., dietary restriction, range of price, and acceptable restaurant rating), user context (e.g., location, available time, and privacy requirements), environmental context (e.g., time, weather, other user reviews, and current traffic), and database-specific context (e.g., for a restaurant we consider current waiting line, opening status, rating, and change of menu).

Towards the goal of realizing a context and preference-aware location-based services, we introduce the system architecture of a preference and context-aware location-based Database Server (*CareDB*, for short), currently under development at University of Minnesota, that delivers *personalized* services to its customers based on the surrounding context. *CareDB* will replace the database server depicted in Figure 1 and will go beyond the traditional scheme of "one size fits all" of existing location-aware database systems. Instead, *CareDB* tailors its functionalities and services based on the context of each customer. System users register their personal preference profiles/context with *CareDB*. In addition, *CareDB* has the ability to collect other data regarding the context of database information and surrounding environment. Upon processing a given query, *CareDB* considers the following: (1) Existing stored data (this is similar to traditional databases), (2) Current user profile that includes user preference and user context (existing systems consider only the location context while ignoring all other context), and (3) Surrounding global context (e.g., time, weather, and road conditions). To show the capabilities of *CareDB*, consider the aforementioned example of restaurant finder application. Before reporting the query answer, *CareDB* will check the user context to get her budget, dietary restriction, and available time, thus would not report expensive restaurant, restaurants with excessive waiting time, nor restaurants with conflicting dietary offerings. In addition, *CareDB* will check the restaurant context in order to know the restaurant current waiting line and its current opening status to be able to report only opening restaurants that have suitable waiting time. Finally, *CareDB* will check the environment context in terms of current road traffic in order to estimate the current travel time to each of the reported restaurants.

Within the framework of *CareDB*, we identify five main challenges that need to be addressed: (1) matching the user profile to the current context, (2) supporting multi-objective query processing as a means of satisfying various forms of user profiles, (3) designing a context-ware query optimizer that takes into account surrounding context when deciding upon the best query plan, (4) designing context-aware query operators that embed context-awareness into the core processing of traditional location-based query operators, and (5) supporting continuous queries that are ubiquitous in context-aware environments.

The rest of this paper is organized as follows: Section 2 covers *CareDB* system architecture. Section 3 discusses five main challenges addressed by *CareDB* with pointers to research directions and solutions. Related is highlighted in Section 4. Finally, Section 5 concludes the paper.

2. SYSTEM ARCHITECTURE

Figure 2 gives the *CareDB* system architecture that is depicted by a bold rectangle. Other than user queries, there are three input types for *CareDB*, namely, the *user preference/context*, the *database-specific context*, and the *environmental context*. Each context has two components, *static* context that is rarely changing (represented by solid lines and dark gray rectangles) and *dynamic* context that is frequently changing (represented by dotted lines and light gray rectangles). Details of the input context are as follows:

- **User preferences/context.** Registered users with *CareDB* have the ability to specify their personal preferences along with their context. For example, a user may specify that whenever she is looking for a restaurant, she would like that the query processor take into account distance, price, rating, and dietary restriction while when looking for gas stations, the user wants to consider only the distance and the preferred gas company. Using such preferences, the preference- and context-aware query processor will consult the appropriate context to provide a context-aware answer that is tailored to both the user preferences and user context. Parts of user context are static, i.e., rarely changing, for example, income, profession, health condition,

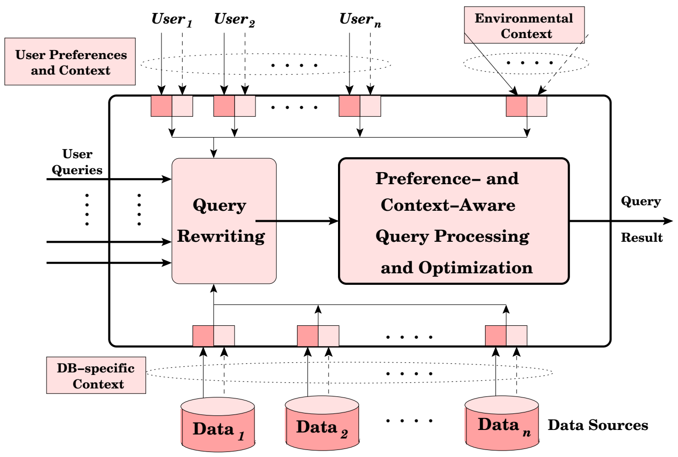


Figure 2: System Architecture

age, and privacy requirements. Other parts of user context can be highly dynamic, for example, user location which is continuously changing based on the user movement.

- **Database-specific context.** Data sources can register with *CareDB* by their context. Examples of data sources include restaurant databases, hotel databases, and taxi databases. Similar to the concept of user context, database-specific context may have static and/or dynamic context types. For example, in a restaurant database, a static context may include meal price, dessert price, cuisine, type of people who like this restaurant, and operating hours while dynamic context may include the current waiting time, the current number of customers, and today's special. In order to find a suitable restaurant for a certain user, the preference- and context-aware location-based query processor would match user preferences and context with the restaurant context.

- **Environmental context.** Unlike user and database context that are stored either at the query issuer side or at the data source side, the environmental context is stored at a *third party* that is consulted by the location-based query processor to enhance the answer quality of location-based queries. Examples of dynamic environmental context include current road traffic (i.e., estimated travel time through road segments), time, weather, and neighboring peers of the query issuer. Examples of static environmental context include third-party ratings of restaurants, gas stations, hotels, or any other third-party statistics about data sources. Examples of queries that get help from the environmental context include a restaurant finder query where the query processor consults the environmental context to get the current traffic and estimated travel time to each restaurant, then, the computed distance is fed to the query to get an accurate answer. Another example that makes use of the data collected by third parties is also a restaurant finder query in which the user wants to choose the best restaurant in terms of other user reviews and scores to each restaurant.

As depicted in Figure 2, at the front end of the system, all user queries go through a *query rewriting module* that receives simple snapshot and continuous location-based queries, investigates what are the context that need to be considered, and adds the context as conjunctive predicates to the issued query. The core part of *CareDB* is the *preference- and context-aware query processing and optimization module*. The main responsibilities of this module are to embed context-awareness into the existing core processing of query operators, support the integration between context-aware queries and a wide class of location-based queries, and provide support of environmental context as well as both the user and database context. It is important to note that the *preference- and context-aware query processing module* does not support new types of location-based queries, instead, it enhances the quality of answer of existing location-based queries by incorporating user preferences, user context, database-specific context, and environmental context into the query answer.

3. CHALLENGES AND RESEARCH DIRECTIONS

This section outlines five main challenges and research directions towards building a context and preference-aware location-based database server, namely, mapping user profiles into context (Section 3.1), multi-objective queries (Section 3.2), context-aware query optimization (Section 3.3), context-aware query operators (Section 3.4), and continuous queries (Section 3.5).

3.1 Challenge I: Preference Profiles and Contexts

CareDB performs query processing with knowledge of user preferences and surrounding contexts. Thus, two main challenges exist in managing this information: (a) Define a user preference profile that reflects user preferences in various dimensions. The more preferences contained in the profile the more accurate and relevant the query answer will be, yet, the more complicated query processing will be. (b) Map any required contextual data to the attributes in the preference profile. It is of essence to the query optimizer to know exactly the set of context data that must be considered for the query.

Figure 3 gives an example of a user preference profile when issuing a restaurant finder query. User profiles are depicted as bullets while matching user context are depicted as text inside dotted rectangles. In that profile, the user explicitly specifies that upon looking for a restaurant, she wants to take into consideration: (a) The restaurant rating that can be divided into general and special ratings based on the user dietary preferences. This attribute relies on both the *user* context (dietary preference) and the *database* context (restaurant rating). (b) The restaurant price for main course and the dessert, that relies on the *database* context (restaurant price). (c) The total time needed to start the meal which can be computed as the travel time from the user location to the restaurant location plus the waiting time at the restaurant. This attribute depends on the three context types, *user* context (the current user location and available time), *database* context (the restaurant location and current average waiting time), and the *environmental* context (the current traffic conditions that can be used to compute the

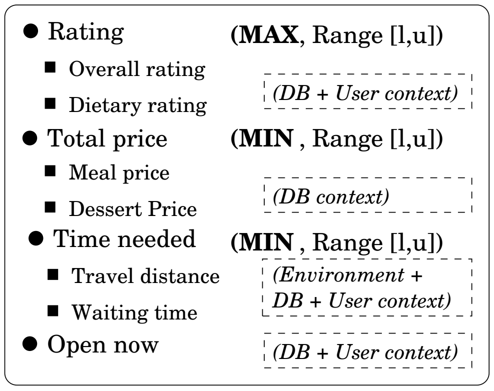


Figure 3: User Preference Profile.

current travel time). (d) Based on the current time, the user may want to look for either opened or closed restaurants. Whenever applicable, the user would specify for each factor whether it needs to be minimized (e.g., travel distance) or maximized (e.g., restaurant rating). Also, a user may specify a range for each factor (e.g., a lower and upper bound for price). In fact, this preference profile can handle more robust preference models, such as PreferenceSQL fuzzy constraints [37] (e.g., Around).

It is important to emphasize that a user may have multiple preference profiles for different time instances and different queries. For example, at the lunch time, the user may turn her profile to ignore the dessert price and lower the time range, or, at non-meal times, the user may ignore the factor of whether the restaurant is currently opening or not. Also, if the user wants to issue a gas station finder query, the user would have different factors to be considered. The *query rewriting module* should be able to take such mapping from user profiles to various context and write the user queries in terms of current preferences and context.

3.2 Challenge II: Multi-Objective Query Processing

With the flexibility in defining customized user profiles, location-based queries tend to be *multi-objective* queries. For example, a restaurant finder query aims to achieve various objectives, e.g., minimum price, closest distance, and highest rating. Such objectives may be contradictory in many cases, e.g., a higher rating restaurant is probably of higher price than that of a lower rating one. Given *multi-objective* queries, it is not immediately clear what answer is "correct". As a result, the correct answer of context-ware location-based queries is *not crisply* defined. For example, consider three restaurants r_1, r_2, and r_3. Each restaurant is represented as a triple (p, d, t) where p indicates the restaurant *price*, d indicates the *distance* to the restaurant, and t indicates the restaurant *rating.* If the three restaurant vectors are $r_1 = (30, 50, 5)$, $r_2 = (40, 60, 4)$, and $r_3 = (35, 20, 3)$, then, it is not immediately clear which of these restaurants could be the right query answer as none of them is clearly cheaper, closer, and has higher rating than all other restaurants. Compare this scenario with the traditional location-based restaurant finder scenario where the right answer is *crisply* defined as the closest restaurant.

In general, there are two extremes for multi-objective queries: *top-k* queries and *skyline* queries. In *top-k* queries, all dimensions (i.e., factors that affect the decision) are considered comparable and can be reduced to only one-dimensional value using a scoring function (e.g., sum or average). Such one dimensional value gives a total order of the query result. Then, the *right* answer can be simply considered as the answer with the highest score. Furthermore, the user may specify a number k to return the top k results as the query answer. On the other hand, in *skyline* queries, all dimensions are considered independent as they cannot be aggregated (e.g., price and distance cannot be aggregated). In this case, the best effort of the query processor is to eliminate those objects that are *dominated* by other objects. For example, if restaurant x is cheaper, closer, and has higher rating than restaurant y, then there is no need to report restaurant y in the answer. By eliminating all points that are dominated by others, the query processor can produce only *skyline* points. In the example mentioned above, the query processor would return r_1 and r_3 only as r_2 is dominated by r_1 (r_1 is cheaper, closer, and has higher rating than that of r_2). Then, it is up to the user to choose the appropriate answer among the set of skyline points.

Practically, neither *top-k* nor *skyline* queries can directly service context-aware applications as some dimensions are dependent and can be aggregated (e.g., meal price and dessert price), while other dimensions are independent and cannot be aggregated, (e.g., rating and distance). Thus, there is a real need for supporting *multi-objective* queries as a compromise between *top-k* and *skyline* queries. In a straightforward way, *multi-objective* queries would perform an aggregation over dependent dimensions followed by a skyline over all independent dimensions. Due to its practicality, it is challenging to find more efficient ways to support *multi-objective* queries. For example, performing the aggregation and skyline operations within one scan rather than treating them as two different independent steps.

In addition, we contend that *CareDB* should be able to handle several other multi-objective preference evaluation methods. Examples of these methods include, but are not limited to, *top-k domination* [39], *k-dominance* [5], or *k-frequency* [6]. These alternative preference methods aim to improve upon *skyline* and *top-k* methods by proposing different criteria defining *how* one data object is preferred to another. The challenge in supporting myriad different preference methods is to create a sustainable implementation inside the database query processor. A possible solution is to implement *each* preference method as a *customized* operator in the database. For *CareDB*, we propose an alternative solution: the creation of a generalized *extensible* query evaluation framework. Such a framework would be implemented inside the query processor, however, unlike a *customized* approach, only the general framework touches the query processor. Meanwhile, each preference method *registers* with the framework through the implementation of extensible functions, that are used by the framework during preference query processing. The advantages of an extensible approach are (1) Sustainability: a generalized approach requires no changes to the query processor when a new preference method is implemented. (2) Flexibility: multiple preference methods can co-exist inside the query processor, and *any* method can be used at query time.

3.3 Challenge III: Context-Aware Query Optimization

In the restaurant finder query with the preference profile

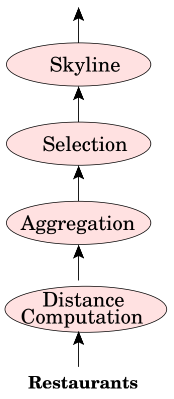


Figure 4: Execution Flow.

given in Figure 3, four types of operations need to take place: (1) *Filters.* The input list of restaurants need to be filtered to get rid of unqualified restaurants, e.g., restaurants that are currently closed or their prices/ratings/distances are out of the user acceptable range. (2) *Aggregation.* Some factors may require performing an aggregation before any decision is taken. Examples of such factors include rating (average all user ratings), price (add meal and dessert prices), and total time needed (add travel time and waiting time). (3) *Extensive computation.* Some factors may require extensive computations before being considered. A typical example is computing the distance from the user location to the restaurant. Such computation requires consulting the environmental context to get the current traffic conditions in order to estimate the shortest route from the user location to the restaurant location. (4) *Skyline computation.* Independent factors (e.g., the total price and total time) need to be entered to a skyline operator to filter out dominated objects that cannot be in the query answer.

Figure 4 gives the execution flow for the four operations in the context-aware restaurant finder query. The first step is to perform an extensive distance computation with all restaurants to estimate the travel time needed to reach the restaurant location. Then, an aggregation will be performed to aggregate the travel time with the average waiting time at the restaurant. Also, the aggregation will be done to get the total price (meal plus dessert) and the average rating (general and dietary rating). Following the aggregation, a filter operation will be performed to get rid of those restaurants that are outside the rating, price, or time specified range. Finally, a skyline operator (or any other multi-objective preference evaluation method) will be employed to get the skyline restaurants in terms of rating, price, and time among the set of qualified restaurants. Notice that if the aggregation gives only value, e.g., all preference factors can be aggregated together to produce one scoring function, then, the *skyline* operator will be replaced by a *top-k* operator.

It is clear that the execution flow in Figure 4 may not be the best possible plan. Thus, it is challenging to design a *context-aware query optimizer* that could accurately estimate the cost and selectivity of each operation to be able to decide upon the best execution flow (i.e., query pipeline). A main objective of the *context-aware query optimizer* is to avoid excessive and redundant computations. Computationally bounded operations should be done only on a request basis. For example, instead of computing the distances to all restaurants, we should aim to only compute the distances for a selective set of restaurants that would have the potential to participate in the final query answer. Such approach would significantly enhance the query processing by avoiding redundant extensive computations. The *context-aware query optimizer* should be able to decide on executing only parts of the query to prune the search before executing more expensive modules. For example, in the restaurant finder query, we may start by doing the aggregation and skyline operations considering only the price and rating attributes. Then, we will compute the distance between user location and restaurants for only those restaurants that appear in the price/rating skyline. The computed distances will provide us with a maximum upper bound of the acceptable restaurants. Then, we can issue a range query to get only those restaurants within the maximum distance.

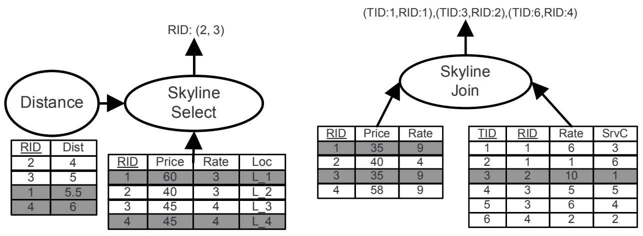
The *context-aware query optimizer* should also take into account sorted attributes and available indexing schemes. For example, consider the case that there exist two copies of the restaurant table where one copy is sorted on price, while the other copy is sorted on rating. In this case, we will start by computing the skyline based on these two fields as skyline computations can make use of the sorted nature of the data to avoid exhaustive data scan. On the other hand, if there is a spatial index for restaurant locations, this would indicate that a range query or nearest-neighbor query can be exploited for search pruning. An important factor that should be taken into consideration by the *context-aware query optimizer* is that traditional attributes (e.g., price and rating) can be pre-sorted while ad-hoc computed attributes (e.g., distance) cannot be pre-sorted as they depend mainly on the *current* user location, i.e., a dynamic user context.

3.4 Challenge IV: Context and Preference-Aware Query Operators

As we discussed in previous sections, preference and context-aware query operators (i.e., *top-k*, *skyline*, and any *multi-objective* operators) need to be developed so that *context-aware query optimization* can take place. In this section, we discuss the non-trivial task of embedding preference and context-awareness into the core processing of traditional query operators. In particular, we focus on *selection*, *aggregate* and *join* operators.

3.4.1 Context and Preference-Aware Selection

A main challenge in adding preference and context-awareness to the selection operator is to perform computationally bounded operations on a request basis. The functionality is best related using an example. Figure 5(a) gives an example of using a skyline selection preference operator, with a single base table of restaurants with three attributes: price, rating, and location. In order to realize the preference function of maximizing rating, while minimizing the price and distance, the location attribute must be translated to distance through extensive computation that consults the current traffic and road network information. To avoid such expensive computations, *CareDB* employs several techniques, we outline two of them: (1) If a nearest-neighbor index is available to compute the distance, we can use it to retrieve restaurants in increasing order based on their distance to the user. Thus, the operator can retrieve the first nearest restaurant, and its distance can be used as a lower



RID	Dist
2	4
3	5
1	5.5
4	6

RID	Price	Rate	Loc
1	60	3	L_1
2	40	3	L_2
3	45	4	L_3
4	45	4	L_4

(a) Select

RID	Price	Rate
1	35	9
2	40	4
3	35	9
4	58	9

TID	RID	Rate	SrvC
1	1	6	3
2	1	1	6
3	2	10	1
4	3	5	5
5	3	6	4
6	4	2	2

(b) Join

Figure 5: Operator Examples

bound to *prune* candidate skylines in that attribute. In Figure 5(a), the restaurant with RID 2 is considered a skyline as it is the closest in the distance dimension. Thus, restaurant RID 1 can be eliminated as it is dominated in the price and rating dimension, and its distance need not be computed as its value is definitely less than the distance of restaurant with RID 2. Similarly, RID 3 is a skyline and RID 4 can be eliminated without a distance computation as it matches RID 3 in price and rating dimensions, but is guaranteed not to be closer than RID 3. (2) If no such distance index is available, the selection operator can compute a sub-skyline using the price and rating attributes, perform distance computations on these select restaurants, and use the largest computed distance in the price/rating skyline as an *upper bound*. A range query can then be issued using the upper bound distance to retrieve all restaurants that can possibly exist in the final answer, and a final skyline evaluation will be executed. This approach avoids distance computations on both (a) restaurants *not* in the price/rating skyline and (b) restaurants *not* within the upper distance range.

3.4.2 Context and Preference-Aware Aggregation

The case of having an aggregate operator followed by a skyline operator is natural. For example, as we have mentioned in Section 3.2, a *multi-objective* operator can be represented as an aggregate operator followed by a skyline operator. As *multi-objective* queries are natural in context-aware environments, it is essential to avoid having two separate operators for aggregation and skyline. Thus, it is challenging to embed context-awareness into the traditional aggregate operators. By doing so, we can avoid multiple scans of the same data sets. For example, the aggregation process requires scanning through the whole data while similar functionality will be needed for the skyline. If we embed the aggregation process within the skyline computation, we will end up avoiding a lot of redundant processing and eliminating several tuples from being considered for the skyline processing.

3.4.3 Context and Preference-Aware Join

In many cases, multi-objective evaluation needs to be performed on attributes in multiple tables, necessitating a join. The join operator is usually an expensive database operator that is non-reductive, i.e., the output size is larger than the input size. Thus, keeping multi-objective preference evaluation (e.g., skyline) and join operations in isolation is inefficient compared to an integrative solution. Consider an example where a user wants to have dinner at a restaurant and then use a taxi to get home. Preferences for this night out could be to minimize dinner price, maximize restaurant rating, maximize the taxi service rating, and minimize the taxi service charge. This example is depicted in Figure 5(b), with the *Restaurant* table on the left and the *Taxi* table on the right. Note that the taxi table stores which restaurants it services by a foreign key, thus a one-to-many join is necessary on the *RID* attribute. If the join and skyline were isolated, then a complete join operation, producing all six results, followed by a skyline operation, would be necessary. In contrast, *CareDB* provides an *integrated* solution using a simple nested loop join with *Restaurant* as the inner relation. The first (RID,TID) pair produced would be (1,1). Then, the next tuple in *Taxi* with (RID,TID) key (2,1) could instantly be discarded, as its rating (Rate) and service charge (SrvC) dimensions are dominated by the previously joined tuple (1,1). Note that the output of the integrated skyline join contains only three tuples. Furthermore, this example shows that the skyline operation cannot simply be pushed before the join to evaluate *local* skylines on each table, followed by a join to compute the final result. The *local* skylines are highlighted in gray in Figure 5(b), however, tuple (TID:6,RID:4) is a final skyline tuple, but not a *local* skyline in the *Restaurant* or *Taxi* relation.

The challenge behind embedding preference and context inside the join operation is to *prune* input tuples that cannot possibly contribute to the preference answer *during* the join operations. In other words, we aim to avoid a straightforward naive query plan, when possible, that involves joining *all* input tuples and *then* performing preference evaluation over the complete join result. We are currently exploring an initial technique that computes a set of *local* preference results for each input table separately. While computing the local preference results for each table, we use the join predicate to exploit the fact that certain local tuples cannot possibly be preference answers when joined with their counterpart tuple in the other table. Thus, we mark the potential of each *local* preference answer to be a global one, i.e., appear at the final result. The main motivation is that the join operators are likely to be non-reductive, i.e., the output size of the join operator is larger than its input size. Thus, early pruning is a way to: (1) reduce the cost of the join and (2) reduce the cost of any final preference evaluation that marks join result tuples as final preference answers. Making the join operator preference-aware would greatly affect the overall query performance.

3.5 Challenge V: Context-Aware Continuous Queries

In context-aware environments, continuous queries are ubiquitous where the surrounding context is continuously changing. As has been indicated in the system architecture (Figure 2), each type of context (i.e., user, database, and environment context) has a dynamic component. Such a dynamic component raises new challenges in supporting context-aware continuous location-based queries. Unlike traditional continuous location-based queries where the query answer can be continuously changing with the movement of query object or objects of interest, in context-aware continuous queries, the query answer may change even if neither the query object nor objects of interest have changed their locations. Such a case can take place if the underlying context has changed. For example, consider the restaurant finder continuous query. Even if the query issuer did not change her location, the answer may continuously change due to the change in traffic conditions (environment context), restaurant waiting time (database context), or user available time (user context). It is challenging to accommodate such frequently changing environment when reporting and updating the query answer.

Another challenge in context-aware continuous queries is to support an *adaptive* query optimizer module. Continuous queries tend to reside at the system for long times (e.g., hours or days). However, the query optimizer can select the best query plan only based on the context at the query submission time. As time goes by, the initial context may change dramatically making the initial query plan suboptimal. Thus, a set of re-optimization techniques should be deployed to continuously monitor the performance of the existing query plan with respect to the currently changing context and/or preferences. In addition, the query optimizer would always look for the optimal plan with the change of context. Once a better plan than the current one is found, then it is up to the query optimizer to decide if a switch to the new plan is needed.

Finally, as in traditional location-based queries, *shared execution* is a key for achieving scalable and efficient execution of large numbers of outstanding continuous queries. However, with context-awareness, there are more chances to explore new kinds of *shared execution*, i.e., sharing the underlying context. For example, if two different users issue context-aware queries that involve distance computation over certain road network, it is crucial to monitor the change of overlapped road segments for these two queries together. Once a change in traffic condition takes place, the change should propagate to the two users. This is in contrast to having each query executed independent of the other. In general, it is challenging to figure out various ways of *shared execution* that can make use of the underlying context.

4. RELATED WORK

Location-based systems. With the explosive growth of location-based services, several systems have been developed to provide database support for location-based queries. These systems include DOMINO [38], SECONDO [16], and PLACE [28, 30]. DOMINO [38] provides several location-based features on top of existing DBMS's including dynamic attributes, a spatial and temporal query language, indexing, and uncertainty management. SECONDO [16] is an extensible database system, built to support a plethora of non-standard applications, e.g., location-based services, through algebra modules. The PLACE server [28, 30] provides the first built-in approach to support location-based services through specialized location-based query operators inside database management systems. *CareDB* distinguishes itself from all of these systems as it is the first to go beyond consideration of only the "location" context. *CareDB* considers user *preferences*, and other types of static and dynamic context that include environment, user, and database context. Furthermore, similar to PLACE, *CareDB* exploits a built-in strategy.

Preference and context in databases. Following several theoretical works for expressing user preferences in database systems [2, 7, 8, 25], recent systems have been developed to include preference and context in databases. Examples of these systems include PREFER [17], PreferenceSQL [21, 37], Personalized queries [22, 23, 24], AbmientDB [14, 36], and contextual database [33, 34]. The PREFER system [17] incorporates preferences into a single weighted ranking function where preferred results are generated by finding pre-computed materialized views whose weight function is similar to the query. PreferenceSQL [21, 37] provides new constructs for expressing preference in SQL, rules for combining preferences in a *cascading* or *pareto-accumulation* manner, and rules for translating PreferenceSQL into traditional SQL queries. Personalized queries [22, 23, 24] model preferences using a degree of interest score, where queries are injected with mandatory and secondary preferences based on this score. The resulting query is built using traditional SQL constructs. AmbientDB [14, 36] provides a middleware layer to integrate myriad multimedia servers with mobile devices in order to provide efficient ad-hoc queries in a dynamic mobile environment. Contextual database [33, 34] focuses on modeling contextual preferences, and integrating context into query definitions. *CareDB* distinguishes itself from all these systems as it: (a) provides a full-fledged realization of preference and context-aware databases, (b) goes beyond preference modeling and query rewriting to address processing preferences and context at the query operator levels, (c) unlike other systems that build personalization and context-management modules on-top of existing relational databases, *CareDB* exploits a built-in approach where the preference and context-awareness is embedded into the core processing of query operators, and (d) *CareDB* is equipped with the necessary modules that support the special characteristics of location-based servers, e.g., continuous queries and dynamic environments.

Context definitions. There have been several definitions of context and context-awareness (e.g., see [4, 12, 32]). Most of these definitions define the context in terms of examples with special emphasis on the location context. Similarly, there have been several definitions of context-aware applications that include various synonymous, e.g., adaptive applications [3], reactive applications [9], responsive applications [13], situated applications [18], contented-sensitive applications [31], and environment directed applications [15]. In this paper, we stick with the most formal definitions given by [11] where context is defined as *"any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves"* while a context-aware system is defined as *"A system that uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task"*.

5. CONCLUSION

In this paper, we have discussed the *rigidness* in current location-based applications that provide services based only on the location context while ignoring various forms of user preferences and surrounding context. To overcome such rigidness, we introduced the system architecture of a Context and Preference-Aware Location-based Database Server (*CareDB*, for short), currently under development at University of Minnesota, that delivers *personalized* services to its customers based on the surrounding context. *CareDB* tailors its functionalities and services based on the context of each customer. We have identified three categories of context that should be taken into account when supporting location-based queries, namely, *user* context (e.g., location, budget, and dietary restrictions), *database-specific* context (e.g., restaurant rating, waiting line, and cuisine), and *environmental* context (e.g., weather and traffic conditions). Within the framework of *CareDB*, we have discussed five main challenges that need to be addressed by the research community in order to have a practical realization of context-aware location-based services. The five challenges are: (1) Designing a user profile and mapping it into context parameters, (2) Supporting multi-objective queries in a practical manner, also going beyond the notion of supporting only top-k and skyline queries, (3) Building context-aware query optimizers that take into consideration the surrounding environment when choosing the best query plan, (4) Building context-aware query operators that embed context-awareness into the core processing of traditional query operators (e.g., aggregate and join operators), and (5) Enabling efficient and scalable execution of context-aware continuous queries that takes into consideration the frequent changes of the underlying context in addition to the frequent changes of query and object locations.

6. REFERENCES

[1] ABI Research. GPS-Enabled Location-Based Services (LBS) Subscribers Will Total 315 Million in Five Years. http://www.abiresearch.com/abiprdisplay.jsp?pressid=731. September, 27, 2006.

[2] R. Agrawal and E. L. Wimmers. A Framework for Expressing and Combining Preferences. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2000.

[3] M. G. Brown. Supporting User Mobility. In *IFIP World Conference on Mobile Communications*, 1996.

[4] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware Applications: From the Laboratory to the Marketplace. *IEEE Personal Communications*, 4(5):58–64, Oct. 1997.

[5] 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.

[6] 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.

[7] J. Chomicki. Querying with Intrinsic Preferences. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2002.

[8] J. Chomicki. Preference Formulas in Relational Queries. *ACM Transactions on Database Systems, TODS*, 28(4):427–466, 2003.

[9] J. R. Cooperstock, K. Tanikoshi, G. Beirne, T. Narine, and W. Buxton. Evolution of a Reactive Environment. In *Proceedings of the International Conference on Human Factors in Computing Systems, CHI*, 1995.

[10] The cellular telecommunication and internet association, ctia. http://www.wow-com.com/.

[11] A. K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[12] A. K. Dey, G. D. Abowd, and A. Wood. CyberDesk: A Framework for Providing Self-Integrating Context-Aware Services. *Knowledge-Based Systems*, 11(1):3–13, 1998.

[13] S. Elrod, G. Hall, R. Costanza, M. Dixon, and J. des Rivières. Responsive Office Environments. *Communications of ACM*, 36(7):84–85, 1993.

[14] L. Feng, P. M. G. Apers, and W. Jonker. Towards Context-Aware Data Management for Ambient Intelligence. In *International Conference of Database and Expert Systems*, 2004.

[15] S. Fickas, G. Kortuem, and Z. Segall. Software Organization for Dynamic and Adaptable Wearable Systems. In *International Symposium on Wearable Computers*, pages 56–63, Oct. 1997.

[16] R. H. Güting, V. T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle. SECONDO: An Extensible DBMS Platform for Research Prototyping and Teaching. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2005.

[17] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2001.

[18] R. Hull, P. Neaves, and J. Bedford-Roberts. Towards Situated Computing. In *International Symposium on Wearable Computers*, 1997.

[19] C. S. Jensen. Database Aspects of Location-based Services. In *Location-based Services*, pages 115–148. Morgan Kaufmann, 2004.

[20] C. S. Jensen, A. Friis-Christensen, T. B. Pedersen, D. Pfoser, S. Saltenis, and N. Tryfona. Location-based Services: A Database Perspective. In *Proceedings of the 8th Scandinavian Research Conference on Geographical Information Science, ScanGIS*, 2001.

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

[22] G. Koutrika and Y. Ioannidis. Constrained Optimalities in Query Personalization. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2005.

[23] G. Koutrika and Y. E. Ioannidis. Personalization of Queries in Database Systems. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2004.

[24] G. Koutrika and Y. E. Ioannidis. Personalized Queries under a Generalized Preference Model. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2005.

[25] M. Lacroix and P. Lavency. Preferences: Putting More Knowledge into Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 1987.

[26] D. L. Lee, M. Zhu, and H. Hu. When Location-based Services Meet Databases. *Mobile Information Systems*, 1(2):81–90, 2005.

[27] MapInfo. http://www.mapinfo.com/.

[28] M. F. Mokbel and W. G. Aref. PLACE: A Scalable Location-aware Database Server for Spatio-temporal Data Streams. *IEEE Data Engineering Bulletin*, 28(3):3–10, Sept. 2005.

[29] M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Towards Scalable Location-aware Services: Requirements and Research Issues. In *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM GIS*, 2003.

[30] M. F. Mokbel, X. Xiong, W. G. Aref, S. Hambrusch, S. Prabhakar, and M. Hammad. PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams (Demo). In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2004.

[31] J. Rekimoto, Y. Ayatsuka, and K. Hayashi. Augment-able Reality: Situated Communication Through Physical and Digital Spaces. In *International Symposium on Wearable Computers*, 1998.

[32] B. N. Schilit, N. I. Adams, and R. Want. Context-Aware Computing Applications. In *Workshop on Mobile Computing Systems and Applications*, 1994.

[33] K. Stefanidis and E. Pitoura. Fast Contextual Preference Scoring of Database Tuples. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2008.

[34] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Adding Context to Preferences. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2007.

[35] TargusInfo. http://www.targusinfo.com/.

[36] A. H. van Bunningen, L. Feng, and P. M. G. Apers. A Context-Aware Preference Model for Database Querying in an Ambient Intelligent Environment. In *International Conference of Database and Expert Systems*, 2006.

[37] G. K. Werner Kießling. Preference SQL - Design, Implementation, Experiences. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2002.

[38] O. Wolfson, A. P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. DOMINO: Databases fOr MovINg Objects tracking (Demo). In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 1999.

[39] 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.