Towards Scalable Location-aware Services:
Requirements and Research Issues*

Mohamed F. Mokbel   Walid G. Aref   Susanne E. Hambrusch   Sunil Prabhakar

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{mokbel,aref,seh,sunil}@cs.purdue.edu

ABSTRACT

The emergence of location-aware services calls for new real time spatio-temporal query processing algorithms that deal with large numbers of mobile objects and queries. Online query response is an important characterization of location-aware services. A delay in the answer to a query gives invalid and obsolete results, simply because moving objects can change their locations before the query responds. To handle large numbers of spatio-temporal queries efficiently, we propose the idea of *sharing* as a means to achieve scalability. In this paper, we introduce several types of sharing in the context of continuous spatio-temporal queries. Examples of sharing in the context of real-time spatio-temporal database systems include sharing the execution, sharing the underlying space, sharing the sliding time windows, and sharing the objects of interest. We demonstrate how sharing can be integrated into query predicates, e.g., selection and spatial join processing. The goal of this paper is to outline research directions and approaches that will lead to scalable and efficient location-aware services.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*spatial databases and GIS.*

General Terms

Design, Performance.

Keywords

Location-aware services, spatio-temporal databases, moving objects.

---

1. INTRODUCTION

Combining the functionality of personal locator technologies, global positioning systems (GPSs), wireless and cellular telephone technologies, and information technologies enables new environments where virtually all objects of interest can determine their locations. These technologies are the foundation for pervasive location-aware environments and services. Such services have the potential to improve the quality of life by adding location-awareness to virtually all objects of interest such as humans, cars, laptops, eyeglasses, canes, desktops, pets, wild animals, bicycles, and buildings.

By enabling an upward link, the data sent from the mobile objects to the servers enables an environment in which objects are aware of the locations of surrounding objects as well as other related information. Applications can range from locating lost or stolen objects, to tracking little children and alerting parents when their children step out of the back yard, or completely automating traffic and vehicle navigation systems.

A location-aware server is characterized by the large number of mobile and stationary objects and comparatively small number of fixed regional servers. Mobile and stationary objects have the ability to issue ad hoc spatio-temporal queries. Spatio-temporal queries are queries where both the objects and the query region may change their locations and/or shapes over time. We distinguish between *snapshot* and *continuous* queries. *Snapshot* queries are queries that can be answered using data that is already collected, either in one of the fixed regional servers or in a large repository server. *Continuous* queries are queries whose responses depend on data progressively accumulating into the servers. A continuous query may either report accumulated results at regular intervals of time or may be triggered to report a result when a certain event happens. A key point in the design of a location-aware server is to provide efficient data structures and query optimization techniques to provide fast query responses. In a location-aware environment, where objects are continuously moving, any delay in query response results in an invalid and an obsolete answer. The ability of having a fast query response in a location-aware server is hindered by two main reasons:

1. **Scalability.** Location-aware environments are characterized by the large number of moving objects, and a large number of issued continuous and snapshot queries. In the context of spatio-temporal databases, several spatio-temporal access methods [22] are pro-

posed that are scalable in terms of the number of moving objects that can be supported (e.g., see [5, 24, 25, 27, 28, 32, 34]). The scalability in terms of the number of spatio-temporal queries and moving objects is addressed in [25]. In addition, each spatio-temporal access method is concerned with a specific type of spatio-temporal queries (e.g., range queries [28, 34], nearest neighbor queries [31, 33], reverse nearest-neighbor queries [4], historical queries [24, 32], NOW queries [25], and future queries [27, 28]). Thus even if a spatio-temporal access method is scalable with respect to a certain query type, there is no guarantee that the location-aware server can be scalable with respect to the wide variety of spatio-temporal queries.

2. **Complexity.** Unlike traditional queries, spatio-temporal queries are both CPU and I/O-intensive. Spatial comparisons may require complex geometric algorithms. Efficient algorithms are needed to avoid unnecessary complex computations.

In this paper, we aim to provide research directions towards achieving scalability in location-aware servers. By scalability, we mean the ability of the location-aware server to provide fast responses to a large number of continuous concurrent spatio-temporal queries of different types. We focus on scalability for continuous queries for two reasons: (1) Continuous queries are natural in the highly dynamic location-aware environments. (2) Snapshot queries can be considered as a special case of continuous queries, if the period of execution of the continuous query is very small. Efficient algorithms to handle continuous queries have been introduced in the context of append-only databases [36], internet queries [17, 18], stream processing [3, 21], and spatio-temporal queries [31, 33]. Scalable approaches to handle multiple concurrent continuous queries mainly focus on triggers [15], and internet queries [9, 19]. Scalability in handling continuous spatio-temporal range queries is addressed in [25]. However, the focus is on a specific instance of range queries, where the query regions are stationary and the queries are interested only in the current locations of moving objects. Up to the authors' knowledge, scalability in handling a variety of continuous spatio-temporal queries has not yet been addressed.

A key technique towards achieving scalability in location-aware servers is *sharing*. Examples of *sharing* in the context of real-time spatio-temporal database systems include sharing the execution of multiple yet similar queries, sharing the underlying space, sharing the sliding time windows, and sharing the objects of interest. The *sharing* concept has been introduced in the context of traditional queries [6, 26, 29], continuous queries in the web environment [8, 9], and continuous streaming queries [7, 14, 21]. In this paper, we focus on sharing as a means of achieving scalability for continuous spatio-temporal queries.

The rest of the paper is organized as follows: Section 2 describes a typical location-aware environment that is designed as part of the *Pervasive Location-Aware Computing Environments* (PLACE) project [2], developed at Purdue University. In Section 3, spatio-temporal queries are classified into different categories. Section 4 introduces the concept of *sharing* in continuous spatio-temporal queries. Spatio-temporal join as the main block for *sharing* is discussed in Section 5 . Finally, Section 6 concludes the paper.
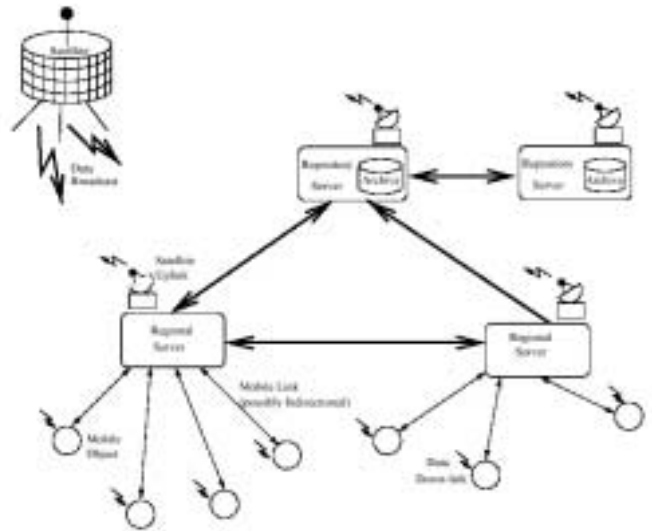

Figure 1: The Pervasive Location-Aware Environment (PLACE) Architecture.

2. THE PLACE PROJECT

In this section, we give an outline of the *Pervasive Location-Aware Computing Environments* (PLACE) project [2], developed at Purdue University. Figure 1 sketches a hierarchical architecture of the location-aware computing environment. Location detection devices (e.g., GPS devices) provide the objects with their geographic locations. Objects connect directly to regional servers that form the lowest level in this hierarchy. Regional servers handle the incoming data and process time-critical spatio-temporal queries. Regional servers communicate with each other, as well as with the high level servers i.e., the repository servers. Repository servers archive the past locations of moving objects.

The servers are interconnected by high bandwidth links. However, the mobile links between the regional servers and objects have low bandwidth and a high cost per connection. For data that is being sent from the moving objects to the regional servers (i.e., uplink direction), we regulate the amount of data collected (*resolution*), and the rate at which it is sent (*upload frequency*). For query results and other data being sent from the location-aware regional servers to the objects (i.e., the downlink direction), an alternative to the point-to-point mobile links is allowed. Servers can transmit data to a satellite that broadcasts the information over the air to all objects. Broadcasting allows a server to send data to a large number of *listening* objects [1, 13].

In traditional applications, GPS devices tend to be passive, i.e., they do not exchange any information with other devices or systems. More recently, GPS devices are becoming active entities that transmit and receive information that is used to affect processing. Examples of these new applications include vehicle tracking, identification of closet emergency vehicles, and Personal Locator Services.

3. SPATIO-TEMPORAL QUERIES

Unlike traditional and spatial queries, in spatio-temporal queries, both objects and query regions may change their

locations and/or shapes over time. Dealing with time results in a wide variety of spatio-temporal queries. In this section, we give classification of a variety of spatio-temporal queries that are supported in the PLACE server.

We give two classifications of spatio-temporal queries that are based only on the time dimension. Thus, the following classifications can be applied to both continuous and snapshot queries. Also, the classifications apply to many types of queries, e.g., range queries, nearest-neighbor queries, and reverse nearest-neighbor queries [4].

The first classification is based on the time of the query. Spatio-temporal queries can ask about information related to the past, current, or future times:

- **Historical Spatio-temporal Queries.** Historical queries ask about the past data (data that is stored at the repository servers in Figure 1). An example of historical queries is *"Find the locations of a certain object between 7 AM and 8 AM today"*. A continuous version of this query is: *"Continuously, Find the locations of a certain object in the last hour"*. In this case, the continuous query time interval (*last hour*) is a sliding time window. To support historical queries, a location-aware server stores only the locations of the moving objects at different times. Once a location of a moving object is updated, the old location is sent to the repository server along with the time the old location was reported. Examples of spatio-temporal access methods that support historical queries include the TB-tree [24], the MV3R-tree [32], and SETI [5].

- **NOW Spatio-temporal Queries.** NOW queries are interested only on the current location of moving objects. An example of a NOW query is *"Based on my current location, what is the nearest gas station?"*. Due to the highly dynamic environment that is supported by location-aware servers, dealing with NOW queries is challenging [10]. To answer NOW queries, a location-aware server keeps track of the latest locations of all moving objects. Examples of spatial access methods that support NOW queries include hashing [30], the VCI-Index [25], the Q-Index [25], and the LUR-tree [16].

- **Future Spatio-temporal Queries.** Future queries are interested in predicting the locations of moving objects. Additional information (e.g., the velocity or destination) need to be sent from the moving objects to the regional servers in Figure 1. An example of a future query is *"Alert me if a non-friendly airplane is going to cross a certain region in the next 30 minutes"*. Notice that in this query, the alert is sent before the actual event happens, hence, is termed a future or predicting query. Examples of spatio-temporal access methods that support future queries include R-tree based structure (e.g., the TPR-tree [28], the R^{EXP}-tree [27], and the TPR*-tree [34]) and quadtree-based structures (e.g. [35]).

The second classification of spatio-temporal queries is based on the mutability of both objects and queries. Spatio-temporal queries allow stationary queries on moving objects, moving queries on stationary objects, and moving queries on moving objects:

- **Stationary Queries on Moving Objects.** In this category, the query regions are stationary, while objects are moving. Example of these queries include *"How many trucks are within the city boundary?"* and *"Find the nearest 100 taxis to a certain hotel"*. In these queries, the query regions (city boundary and hotel neighborhood) are fixed, while the objects of interest (trucks and cars) are moving. Two approaches can be used to support continuous *fixed* queries. The first approach is to index the moving object with a spatio-temporal access method [27, 28, 34]. The second approach is to index the *fixed* queries with a spatial access methods [25, 37].

- **Moving Queries on Stationary Objects.** In this category, query regions are moving, while objects are stationary. An example of this category is *"As I am moving in a certain trajectory, show me all gas stations within 3 miles of my location"*. This category of queries employ traditional methods to organize the fixed objects (e.g., fractals [11] or R-trees [12]). Efficient algorithms that utilize the R-tree are proposed for the continuous single nearest-neighbor queries [33] and the continuous K-nearest neighbor queries [31].

- **Moving Queries on Moving Objects.** In this category, both query regions and objects are moving. An example of such queries is *"As I (the sheriff) am moving in the space, make sure that the number of police cars within 3 miles of my location is more than a certain threshold"*. In this case, the query region is moving. Also, the objects of interest (police cars) are moving. To support moving queries in a location-aware server, moving objects need to be indexed using a TPR-tree like structure (e.g., [27, 28, 34]). Then, special algorithms are developed to process moving queries in TPR-tree-like structures.

4. USE OF SHARING TO ACHIEVE SCALABILITY

As discussed in Section 3, numerous algorithms and access methods are proposed for each query category. Scalability is maintained via efficient implementation of specific algorithms for each category of spatio-temporal queries. In the PLACE project, we go beyond the idea of developing scalable specific algorithms for certain query types. Instead, we aim to provide a scalable framework to support any kind of concurrent continuous spatio-temporal queries. Mainly, we use the *sharing* concept as a scalability solution. Sharing can be achieved by efficiently grouping spatio-temporal queries into different groups. Figure 2 sketches the main design of the query processor in the PLACE server. Continuous queries are registered into the PLACE server, and is passed to the *Query Parser* module. The main functionality of the parser is to exploit the structure and requirements of continuous queries. The output from the query parser is an intermediate format of the continuous query that represents its structure, signature, and requirements. Based on this intermediate format, the *Query Optimizer* classifies the continuous queries into one of M groups. As will be discussed later, spatio-temporal queries within each group can exploit some kind of sharing.
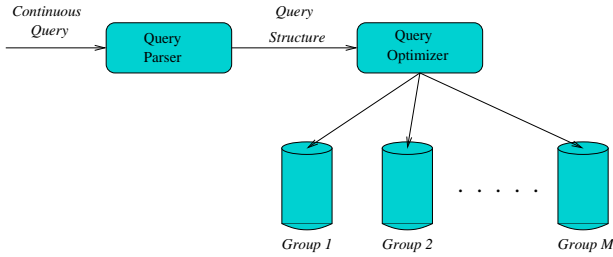
Figure 2: A sketch of the scalable query processor in a location-aware server.

(a) Two range queries with containment property   (b) Decision tree when evaluating Q1 first   (c) Decision tree when evaluating Q2 first

Figure 3: Example of sharing the space in containment queries.

The *sharing* concept is introduced in the context of traditional databases to provide an optimal global plan for a set of multiple concurrent queries with common sub-expressions [6, 26, 29]. Common sub-expressions are evaluated once, where their output is shared among different queries. However, such an approach is limited in three aspects: (1) This approach is not scalable, where it is designed for a limited number of queries. (2) All queries should be known apriori. To optimize for new queries, the whole optimization plan needs to be recomputed. (3) Multi-query optimization deals with snapshot queries only. The concept of sharing as a solution of scalability for continuous queries is introduced in the NiagaraCQ project [8, 9]. The main idea is to apply the *sharing* between queries that have similar query signatures. Thus, *sharing* is not limited to common sub-expressions, but is extended to common computations. New continuous queries can be added to the already existing groups. In the context of data streams, *sharing* is introduced as a scalable solution for continuous queries over streaming data [7, 14, 21]. Due to the unbounded nature of streams, queries over streams are often defined in terms of sliding windows. Thus, sharing in continuous stream queries focus on sharing the computations for different sliding windows [14].

In the PLACE project, we aim to extend the concept of *sharing* to support continuous spatio-temporal queries. Unlike traditional queries that are supported by the NiagaraCQ project [9] and streaming queries that are supported by the PSoup project [7], spatio-temporal queries introduce new kinds of *sharing*. In the following sections, we introduce several kinds of sharing along with the challenges that need to be addressed to efficiently support each kind of sharing.

- **Example:** For the rest of this section, we use the following example as a way to demonstrate our ideas about sharing: Assume that in a vehicle navigation system, a location-aware server keeps track of all moving vehicles in a certain state. The information on moving vehicles is stored in the table *vehicles(ID,location,time,speed,type)* where *ID* is the vehicle identifier, *location* is the last reported location of the vehicle, *time* is the last time that the vehicle reports its location to the server, *speed* is the vehicle velocity, and *type* indicates whether the vehicle is car, truck, or motorcycle. For simplicity, we assume that all queries are continuous and ask about the current time (NOW queries). No future information is stored. However, the same concepts and challenges can be generalized to future and historical queries.
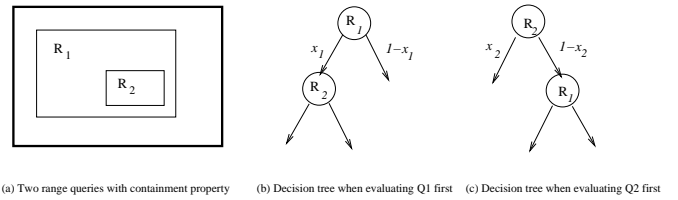
4.1 Sharing the Space

Based on the spatial properties, spatio-temporal queries can share the underlying spatial domain. Two spatio-temporal range queries are considered to have the *Spatial Containment* property if one of them is included in the other. For example, two spatio-temporal range queries with regions R_1 and R_2, where $R_1 \subset R_2$, have the *Spatial Containment* property. In this case, we say that the first query is *spatially contained* in the second query. From the *sharing* prospective, both queries share the area R_1. Thus, the area R_1 needs to be evaluated once, where the answer is utilized by both queries. The *spatial containment* property can greatly reduce the amount of computation needed by a location-aware server. The two previous range queries with regions R_1 and R_2 can be evaluated with only one comparison. For example, If a certain *vehicle* is determined to be inside region R_1, then, we know that it also inside R_2. Similarly, if we know that a certain vehicle is outside R_2, then, we know also, that this vehicle is outside R_1. However, some cases still need two comparisons. For example, if an object is determined that it does not lie inside R_1, then a second comparison with R_2 is needed.

Spatio-temporal queries that have the *spatial containment* property are very frequent in location-aware servers. For example, consider the following two queries: *"How many vehicles are inside the city of West Lafayette?"* and *"How many vehicles are inside the Purdue University campus?"*. Since the Purdue University campus is located inside West Lafayette, the second query is *spatially contained* in the first query.

The sequence of evaluating *spatially contained* range queries is challenging. Even the simple case where there exist only two range queries is not trivial. Cost models need to be provided to decide the optimal sequence of evaluating *spatially contained* spatio-temporal queries. Figure 3a gives an example of two range queries R_1, R_2, where $R_2 \subset R_1$. The bold rectangle is Figure 3a indicates the universe space. Assuming that the distribution of moving objects in the universe is uniform, and that the area of the universe space is one, then the probability that a certain object lies inside region R_i is the area of R_i. We assume that the probability that a certain object lies inside R_1 and R_2 is x_1 and x_2, respectively ($x_2 < x_1$). Figure 3b gives the sequence of evaluating R_1 before R_2 in a *Decision Tree* format. Each circle represents a query evaluation, the left link indicates the sequence when the query is satisfied, while the right link indicates the sequence when the query is not satisfied. A label on any link indicates the probability that this link is used in the evaluation plan.

In Figure 3b, the two queries can be evaluated using only one comparison with probability $(1 - x_1)$ (i.e., if a moving

object is not inside R_1, then it is definitely not inside R_2). However, two comparisons may be needed with probability x_1 (i.e., if a moving object is inside R_1, then we need to test whether it is inside R_2 or not). The expected number of comparisons in this plan is $E_1 = (2)(x_1) + (1)(1 - x_1) = 1 + x_1$. Figure 3c gives the counter plan for evaluating R_2 before R_1. The expected number of comparisons in this case is $E_2 = (1)(x_2) + (1 - x_2)(2) = 2 - x_2$. Comparing the expected number of comparisons yields the following results:

1. If $x_1 < 1 - x_2$, then executing R_1 first is optimal as in Figure 3b.

2. If $x_1 > 1 - x_2$, then executing R_2 first is optimal as in Figure 3c.

3. If $x_1 = 1 - x_2$, then both query plans are equivalent where they result in the same expected number of comparisons.

To illustrate with a numerical example, assume that $x_1 = 0.7$ and $x_2 = 0.1$. Then, $E_1 = 1.7$, while $E_2 = 1.9$. In this case, evaluating R_1 first would be optimal with an average 1.7 comparisons for each moving object. On the other side, if $x_1 = 0.8$ and $x_2 = 0.5$, then $E_1 = 1.8$, while $E_2 = 1.5$. In this case, evaluating R_2 first would be optimal with average number of comparisons 1.5. Notice that organizing R_1 and R_2 inside an R-tree-like structure always requiring two comparisons to evaluate R_1 and R_2. The main reason is that most likely that both R_1 and R_2 will lie in the same R-tree node. Thus, the space *sharing* technique can be applied within an R-tree node, as well as combined with other *sharing* techniques.

Utilizing the *spatial containment* property for continuous spatio-temporal queries is challenging. In the following, we point out some challenges in utilizing the *spatial containment* property to support continuous spatio-temporal queries:

1. In Figure 3, we illustrate the alternative query plans for only two range queries with *spatial containment* property. Generalizing the idea to multiple *spatially contained* queries is challenging.

2. Similar cost model can be used in case of non-overlapping queries; two range queries R_1 and R_2, where $R_1 \cap R_2 = \phi$. This case is trivial, where evaluating the larger query first is always better. Notice that if a certain object is determined to lie inside R_2, then it definitely does not lie inside R_1. Thus, the two queries are evaluated with only one comparison. However, if the moving object does not lie in R_2, we still need another comparison with R_1. The generalization of multiple non-overlapping queries is straightforward. Range queries are sorted based on their areas; queries with large areas are evaluated first. Although this case is trivial, it lays out the foundation for extending the idea of *sharing* the space into overlapped queries.

3. The same concept of sharing the space can be extended to overlapping queries that *share* some part of the space. We consider two range queries with areas R_1 and R_2 are overlapping if $R_1 \cap R_2 \neq \phi$, and there is no *spatial containment* between R_1 and R_2. Figure 4a gives three different range queries with regions R_1, R_2,

(a) Three range queries   (b) The optimal decision tree
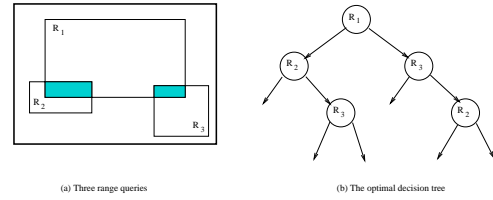
Figure 4: Example of sharing the space in ovrelapping queries.

and R_3. The optimal query plan in a decision tree format is given in Figure 4b. The main idea is to start by evaluating R_1 where R_1 has the largest area. If a moving object is determined to be inside R_1, then, we deal with R_1 as if it is the universe. In this case, we consider only the areas from R_2 and R_3 that are overlapped with R_1 (the gray area). Since the gray area of R_2 is larger than the gray area of R_3, then we evaluate R_2 first then, R_3. On the other hand if a moving object is determined that it is not inside R_1, then we consider only the white areas of R_2 and R_3. In this case, R_3 need to be evaluated before R_2. Notice that the order of evaluating R_2 and R_3 depends on the result of evaluating R_1. Systematic or heuristic methods for building such decision trees need to be developed to achieve a scalable location-aware server.

4.2 Sharing the Query Operator

Basically all queries that query the *Vehicles* table share the same underlying structure (e.g., the *Vehicles* Table). This kind of *sharing* is common to all environments and is exploited in details in [8, 9]. However, as we will see below, several challenges need to be addressed in the context of spatio-temporal queries. Examples of queries that share the same underlying structure include: *"Continuously, how many vehicles are in region R_1?"*, and *"Alert me if the number of vehicles in region R_2 is above a certain threshold from now onwards"*. These two queries share the following SQL part:

SELECT   V.ID
FROM   Vehicles V
WHERE V.location *inside* R_i

where *"inside"* is a spatial operator that checks whether a point lies inside a rectangular region R_i or not. A typical query evaluation plan for each query is: (1) read the *Vehicle* table, (2) Compare each vehicle location to the query region using a point-rectangle inclusion test. Figure 5a gives the query evaluation plan for two range queries with different regions R_1 and R_2. The *Table Scan* operation is repeated twice for the two queries. This can be generalized if a location-aware server receives N different range queries with N different regions $(R_1, R_2, \cdots, R_N)$. Then, the expensive *Table Scan* operation is repeated N times. In a location-aware environment where there is a large number of moving objects, the *Table Scan* operation becomes an I/O-intensive operation.

By utilizing the concept of *sharing*, the query regions can be stored in the *Region(RID,Extents)* table, where *RID* is the query identifier, and *Extents* is the rectangular query
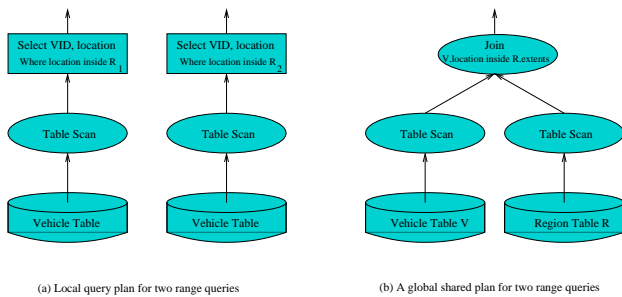
(a) Local query plan for two range queries   (b) A global shared plan for two range queries

Figure 5: An example of sharing a global plan between two range queries.

Figure 6: Queries that share the same interest.

region. In this case, the *Vehicles* table can be read only once, and joined with the *Region* table as in Figure 5b. The output of the join is the tuples *(VID,location,RID)*, indicating that a vehicle *VID* satisfies the query region *RID*. New continuous range queries received by the parser in Figure 2 are translated into *Insert* queries that insert a new record in the *Region* table with new *RID*, and the *Extents* as the query region.

To justify using the shared global plan, we need to make sure that the cost of the join operation between the *Vehicles* table and the *Region* table is less than the cost of reading the *Vehicles* table N times. For example, if a non-indexed nested loop join is used where the outer relation is the *Region* table, while the *Vehicles* table is the inner relation, this would be equivalent to handling each query individually. Thus, an efficient implementation of the join operation is needed. This can be achieved as follows:

- For *Stationary Queries*, where only the objects are moving, it is better to build an R-tree-like spatial index on query regions [25]. Then, an indexed nested loop join is used where the *Vehicles* table is the outer relation, while the inner table is the *Region* table. The standard R-tree point-in-rectangle query is used to find which vehicles satisfies which queries. Notice that in such highly dynamic environment, it is not practical to build an index on moving objects. A spatial index on moving objects cannot afford the high frequency of updating object locations. Notice that moving objects update only their current location information and do not send their velocity information. Thus, a TPR-like structure cannot be used in this case.

- For *Moving Queries on Stationary Objects*, it is better to build an R-tree like index for points locations. Then, an indexed nested loop join is used with the *Region* table is the outer relation. The standard range query algorithm for R-tree is used.

- For the case of *Moving Queries on Moving Objects*, having an index on any of the tables would not help in such a highly dynamic environment. Thus, the problem is transformed to a non-indexed spatial join problem, where one of the relations contain spatial objects with no extents (i.e., points), while the other relation contains rectangular objects. A variation of the spatial hash join [20, 23] can be used to efficiently perform the join operation.
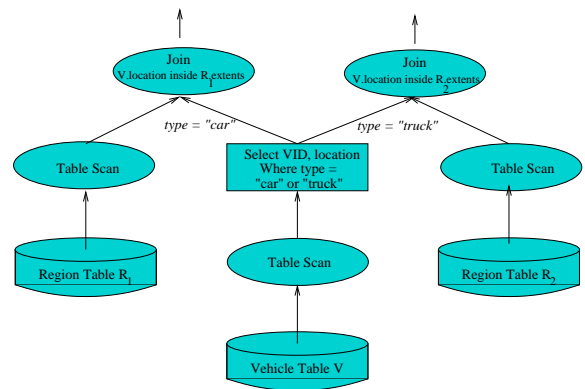
4.3 Sharing the Object Interest

Some spatio-temporal queries may be interested in specific types of objects. For example, consider the following continuous queries: *"How many cars are in Purdue Campus from now onwards?"* and *"How many trucks are in Purdue Campus from now onwards?"*. Although the two queries share the underlying structure (e.g., the *Vehicles* table), however the two queries have different interests. The first query is interested only in vehicles with type *"car"*, while the second query is interested in vehicles with type *"truck"*. These two queries share the following SQL part:

SELECT   Count(V.ID)
FROM   Vehicles V
WHERE   V.type = *type*
AND   V.location *inside R*

Applying the shared global query evaluation plan given in Figure 5b results in unnecessary joined tuples. Queries that have interest only in cars may be joined with moving trucks. Similarly, queries that have interest only in trucks may be joined with moving cars. Figure 6 gives a query evaluation plan for spatio-temporal queries that share the same interest. A *Region(RID,Extents)* table is constructed for each object of interest. In Figure 6, we have two *Region* tables one for cars and the other for trucks. The *Vehicle* table is read once. Based on the selection predicate, tuples are forwarded to one of the join operators. Thus, the join is performed only between queries and objects of the same interest. Notice that the selection condition is performed before the join. This approach is termed as the *PushDown* approach [8], where the selection operator is pushed below the join operator.

Research issues in this kind of sharing include: (1) Developing a cost model that justifies pushing the selection down, rather than performing the join then having the selection. (2) Study the trade-offs of keeping two separate *Region* tables rather than having only one *Region* table followed by a selection operator. (3) The selectivity estimation should be employed to decide whether it is worth to have a separate table for a specific object type. (4) Object types can be grouped into categories, with one table per group rather than one table per type.

4.4 Sharing the Selection

In this kind of sharing, spatio-temporal queries are interested in an attribute that can have continuous values (e.g., the *speed* attribute in the *Vehicles* table). For example, consider the following continuous queries: *"Continuously, find all vehicles with speed greater than 30 in Purdue Campus"* and *"How many vehicles with speed greater than 40 in Purdue Campus from now onwards?"*. Both queries share some results. A vehicle with speed 50 satisfies both queries. These two queries share the following SQL part:

```
SELECT   V.ID
FROM     Vehicles V
WHERE    V.speed > speed
AND      V.location inside R
```

Applying the shared global plan in Figure 5b results in unnecessary joins. For example, vehicles with speed 20 will be joined with the two queries. In addition, vehicles with speed 35 is joined with queries that are interested in speed over 40. Also, applying the plan in Figure 6 results in inefficient performance. Although, there will not be any unnecessary joins, many redundant tuple joins will be performed. For example, consider the case where most of the objects have speed above 40. These objects will be joined twice with each query table. Since both queries have the same space domain (Purdue campus), then the join operation is redundantly repeated.
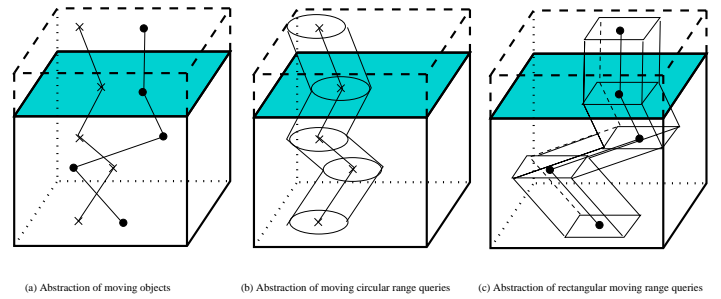
A better approach is to use the *Filter PullUP* approach [8]. The main idea is to pull up the selection predicates after the join operator. Thus, the *Vehicles* table is joined with the *Region* table, then the selection criteria is applied. In this case, redundant joins are avoided. However, there is a chance of unnecessary joins that can result from moving vehicles with speed below 30. To avoid these unnecessary joins, a *filter* is used. The *filter* is another selection operator that is inserted before the join. The selection condition is the union of all the selections. In this case, the filter is to select vehicles with speed above 30. In this approach, there is still a chance of unnecessary join for vehicles with speed that is between 30 and 40. However, the cost of these unnecessary joins is amortized by the gain from sharing the selection operator.

4.5 Sharing the window join

This kind of sharing is very relevant to continuous stream queries [7, 14, 21]. The main idea is to share the join operation among different moving objects. Since objects are moving continuously, then the join is defined in terms of a sliding window. Examples of these queries include: *"Continuously, find the cars that passed by the same location that a truck passed by in the last hour"* and *"Continuously, find the cars and trucks with the same speed in the last 10 minutes"*. These queries deal with the moving objects as a stream of spatio-temporal data. The shared part in these queries is the sliding time window. Joins need to performed in a way that share the execution of the inner sliding window [14].

5. SPATIO-TEMPORAL JOIN

As discussed in the previous section, most of the sharing techniques require a join between the objects table and the queries table. The contents of at least one of these tables

(a) Abstraction of moving objects   (b) Abstraction of moving circular range queries   (c) Abstraction of rectangular moving range queries

Figure 7: The abstraction of moving objects and moving queries.

is continuously changing over time. An efficient implementation of spatio-temporal join is a key point in supporting *sharing* as a means of scalable location-aware servers.

On the abstract level, spatio-temporal join can be considered as a spatial join. Figure 7 gives the abstraction of moving objects, moving circular range queries, and moving rectangular range queries. The current time is represented by the shaded area. The solid hyper rectangle represents the historical data, while the dashed hyper rectangle represents the future predicted movement. Moving objects (Figure 7a) are abstracted as points in the current time, lines in the future time, and trajectories in the past time. Moving circular range queries are abstracted as circle, cylinder, and a sequence of cylinders in the current, future, and past times, respectively. Similarly, rectangular range queries are abstracted as rectangles, hyper-rectangles, and sequence of hyper-rectangles in the current, future, and past times, respectively.

For NOW continuous queries (e.g., queries that ask about the current location), the spatio-temporal join in the context of *sharing* is abstracted as a spatial join where one of the relations contains points (objects with no extents), while the other relation contains rectangles. Due to the highly dynamic environment, it will be difficult to maintain spatial indicies. Thus, a spatial index algorithm that does not utilize any index on the relations can be employed (e.g., see [20, 23]). Future queries are abstracted as a spatial join where one of the relations contains lines, while the other relation contains hyper-rectangles. Again, a non-index-based spatial join algorithm can be utilized. Similarly, the historical queries can be abstracted as a spatial join. However, spatial joins in historical queries are complicated due to the complexity of object and query representations.

As an alternative to the abstraction approach, we can use spatio-temporal join algorithms that utilize the existing spatio-temporal access methods. However, there is no much research in this area. For example, spatio-temporal join algorithms need to be developed for the TB-tree [24] to support historical queries, and for the TPR-tree [28] to support future queries.

6. CONCLUSIONS

Location-aware services are characterized by the large number of mobile and stationary objects, and by the large number of continuous concurrent queries. In this paper, we address the problem of scalability of location-aware services as part of the *Pervasive Location-Aware Computing*

Environments (PLACE) project at Purdue University. By scalability, we mean the ability of a location-aware server to provide real-time responses to a large number of continuous concurrent spatio-temporal queries. We propose the *sharing* concept as a means for achieving scalability in location-aware servers. Several types of *sharing* are introduced that are specific to the spatio-temporal domain. Location-aware servers can also utilize the state-of-the-art *sharing* concepts from other environments (e.g., web continuous queries [9] and streaming continuous queries [7]). Research issues and challenges toward achieving a scalable location-aware server are discussed.

7. REFERENCES

[1] S. Acharya, M. J. Franklin, and S. B. Zdonik. Disseminating Updates on Broadcast Disks. In *VLDB*, pages 354–365, Bombay, India, Sept. 1996.

[2] W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Pervasive Location Aware Computing Environments (PLACE). http://www.cs.purdue.edu/place/.

[3] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109–120, 2001.

[4] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *Proceedings of the International Database Engineering and Applications Symposium, IDEAS*, pages 44–53, Alberta, Canada, July 2002.

[5] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets with SETI. In *Proc. of the Conf. on Innovative Data Systems Research, CIDR*, Asilomar, CA, Jan. 2003.

[6] U. S. Chakravarthy and J. Minker. Multiple Query Processing in Deductive Databases using Query Graphs. In *VLDB*, pages 384–391, Kyoto, Japan, 1986.

[7] S. Chandrasekaran and M. J. Franklin. Streaming Queries over Streaming Data. In *VLDB*, pages 203–214, Hong Kong, 2002.

[8] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *ICDE*, San Jose, CA, 2002.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, pages 379–390, 2000.

[10] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of "Now" in Databases. *ACM Transactions on Database Systems , TODS*, 22(2), 1997.

[11] C. Faloutsos and S. Roseman. Fractals for Secondary Key Retrieval. In *PODS*, pages 247–252, Philadelphia, PA, Mar. 1989.

[12] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57, Boston, MA, June 1984.

[13] S. E. Hambrusch, C.-M. Liu, W. G. Aref, and S. Prabhakar. Query Processing in Broadcasted Spatial Index Trees. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, pages 502–521, Redondo Beach, CA, 2001.

[14] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, Berlin, Germany, Sept. 2003.

[15] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable Trigger Processing. In *ICDE*, pages 266–275, Sydney, Austrialia, Mar. 1999.

[16] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile Data Management, MDM*, pages 113–120, Jan. 2002.

[17] L. Liu, C. Pu, R. S. Barga, and T. Zhou. Differential Evaluation of Continual Queries. In *Proceedings of the International Conference on Distributed Computing Systems, ICDCS*, pages 458–465, Hong Kong, May 1996.

[18] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 11(4):610–628, 1999.

[19] L. Liu, C. Pu, W. Tang, D. Buttler, J. Biggs, T. Zhou, P. Benninghoff, W. Han, and F. Yu. CQ: A Personalized Update Monitoring Toolkit. In *SIGMOD*, pages 547–549, Seattle, WA, June 1998.

[20] M.-L. Lo and C. V. Ravishankar. Spatial Hash-Joins. In *SIGMOD*, pages 247–258, Montreal, Canada, 1996.

[21] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, Madison, Wisconsin, June 2002.

[22] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, June 2003.

[23] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD*, pages 259–270, Montreal, Canada, 1996.

[24] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *VLDB*, pages 395–406, Cairo, Egypt, Sept. 2000.

[25] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[26] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*, pages 249–260, 2000.

[27] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE*, San Jose, CA, Feb. 2002.

[28] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, pages 331–342, May 2000.

[29] T. K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems , TODS*, 13(1):23–52, 1988.

[30] Z. Song and N. Roussopoulos. Hashing Moving Objects. In *Mobile Data Management*, pages 161–172, Hong Kong, Jan. 2001.

[31] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, pages 79–96, Redondo Beach, CA, 2001.

[32] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB*, pages 431–440, Rome, Italy, Sept. 2001.

[33] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *VLDB*, pages 287–298, Hong Kong, 2002.

[34] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*, Berlin, Germany, Sept. 2003.

[35] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3):185–200, 1998.

[36] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*, pages 321–330, San Diego, CA, 1992.

[37] Y. Xia and S. Prabhakar. Q+Rtree: Efficient Indexing for Moving Object Database. In *Proceedings of the International Conference on Database Systems for Advanced Applications, DASFAA*, pages 175–182, Kyoto, Japan, Mar. 2003.