Continuous Queries on Current and Predicted Locations of Moving Objects

Group d405a
Computer Science Department
Aalborg University
Laurynas Siksnys, Katja Springer, Simon Nicholas M. Tinggaard, Rune L. Wejdling
{siksnys, katja, nicholas, runelw}@cs.aau.dk

May 30, 2008

Abstract

In recent years there has been an increase in the number of advanced location based services (LBS). When the number of users becomes large, all LBS location servers suffer from huge amounts of updates. Recently developed location update policies, like so-called velocity-vector-based update policy, in comparison to currently used methods, achieve less location updates with the same given accuracy. There is a challenge to integrate this update technique into a LBS system and thereby significantly reduce communication traffic between clients and the server.

We present a general purpose database solution, which can be used as a core element in most LBS systems. It employs spatial indexing on continuous queries, while using velocity-vector-based shared prediction update technique to provide an efficient evaluation of spatio-temporal queries on moving objects. We present an implementation of our solution on a spatially enabled relational database system and provide performance study results.

1 Introduction

In the recent years, there has been a large increase in the amount of location aware devices like car navigators and mobile phones. If these devices are provided with an Internet connection, they can connect to on-line services that can service the user of the device with, e.g., information on nearby restaurants or cultural offers. By letting the location-aware device commit location information to the on-line service, the user can avoid entering location information manually which can be an inconvenience. Other on-line service applications could track users movement by letting their location-aware devices commit location information on a regular basis, enabling their friends to find them at any given time [4].

Knowing the current movement behaviours of objects like cars, in addition to its current location, can enable a system to predict which objects will be in a given area at a given time in the future. Service stations like, e.g., roadside restaurants, are interested in knowing how many possible customers that are or will be in the area to make the necessary preparations. By utilising positioning data from, e.g., cars the restaurant can make a qualified guess on which customers will be in the area a given time in the near future.

Future object location predictions become possible when applying velocity-vector-based update policies [2] in order to minimize location-aware device and server communication costs. In many cases, communication traffic reduction provides cheaper service to the user and helps saving battery power on the user's mobile device. This is achieved by continuously performing moving object location prediction both - in the location-aware device and on the server. The position of a moving object is only updated when the deviation between the predicted location and the actual position of the object exceeds some given accuracy.

Efficient querying techniques have been proposed [8] [6] [5] for providing the past, current or future locations of objects, given a large set of objects. Some of the techniques achieve good performance on random spatio-temporal queries, but suffer from a large number of updates generated by changes in location of objects. Other techniques work with continuous queries in order to provide rapid response to the user by maintaining results of each query through its lifetime. Although continuous queries may be expensive to insert, update and delete, they are applicable in many real-life applications. Different models of software systems, that support continuous queries for multiple representations of object movement behaviours, have been presented [6] [5]. Some systems represent moving objects by stationary points, that represent their locations only for limited periods of time, others assume that object is somewhere in a circular area (not further than some fixed distance from the object's last reported location). None of these representations work with velocity-vector-based shared prediction update policies. Enabling spatio-temporal queries for moving objects in a system, that employs velocity-vector-based shared prediction update policy, poses new data storage, indexing, and processing challenges. In this work we assume that each object moves with fixed speed in a fixed direction for a period of time and present a solution for a system that supports spatio-temporal queries together with a velocity-vector-based shared prediction update policy.

The basic model of a system contains a database server interacting with query subscribers and moving object clients. There is a big number of clients committing their current positioning data to the database server. The database server enables query subscribers to subscribe to spatio-temporal queries. The subscription enables query subscribers to execute the query periodically. Each query returns a set of objects that satisfy the given space and time constraints in an acceptable time for real-time applications. For simplicity, space and time are constrained by static rectangular areas and time intervals.

We propose a solution that can handle a large number of moving object clients and continuous query subscribers. A prototype of this solution is implemented and tested in a commercial relational database management system (RDBMS).

The remainder of this paper is structured as follows. Section 2 covers related work, followed by a detailed description of problem being addressed. Section 4 describes our proposed solution and possible optimisation strategies are presented in Section 5. The solution is applied in Section 6, where performance study results on a prototype system are provided.

2 Related work

The problem of continuous querying on moving objects is widely discussed and multiple solutions have been proposed [5] [8] [7] [6]. Multiple ways are found to solve a query like:

"*Continuously determine the set of objects that are within the query*". We extend the query to predict the movement of the objects by utilising the velocity vector. A query would then be: "*Continuously determine the set of objects that will be within the query in 3 to 5 minutes*".

A naive *brute force* approach, is to compare each object against the query when the query is executed, to produce a set of objects that match the given query. Building a spatial index on the objects speeds up the execution time of the query. When the number of objects increases, so does the number of updates to the system, increasing the number of updates to the spatial index, which has shown to be very costly [8]. To avoid this scenario we build a spatial index on the queries, as introduced in [5] and [6]. It is generally assumed that updates to the query ranges are less frequent than location updates from the moving objects. By creating the spatial index on the queries, the number of updates to the spatial index can be decreased, since the index is only updated when new queries are added or modified. The spatial index is used when an object reports a new position, to identify the queries that match the new position information.

By utilising a shared prediction policy technique called vector-based tracking, presented in [2], we can reduce the number of updates made by the moving objects.

3 Problem Description

The goal is to develop the previously described database server solution for tracking of moving objects and query evaluation that would operate in a setting with following assumptions and non-functional requirements:

- The number of moving objects is typically higher than the number of queries.
- The frequency of query executions and moving object location updates is significantly higher than the frequency of insertions and deletions of objects or queries.
- Single query spatial constraint is constant, while temporal constraint is changing. Queries with different *spatial* constraints must be explicitly registered to the system in advance, however concrete dynamic *temporal* constraints are specified on each query operation call.
- It is enough to provide query result only for bounded time window in range between current query execution time and some d_{max} time units into the future. It is required that d_{max} is either a finite number or some number close to infinity.
- The number of analysed objects during each query evaluation should be minimised in respect to a brute force method.

The database server solution must support entities, for which the formal definitions are presented below. Moving object $o \in \mathbf{O}$, where $\mathbf{O}$ is the set of all possible objects, is interpreted as continuously moving point in 2-dimensional space with fixed speed and direction. It can be defined by a 4-tuple $o = (oid, \vec{loc_{obs}}, \vec{v_{obs}}, t_{obs})$, where $oid \in \mathbb{N}$ is an unique object identification value, $\vec{loc_{obs}}$ is observed location, $\vec{v_{obs}}$ is observed velocity vector and t_{obs} is observation time. The predicted location of moving object o can be computed for any time $t \geq t_{obs}(o)$ by $\vec{loc}(o,t) = o.\vec{loc_{obs}} + o.\vec{v}_{obs} \cdot (t - o.t_{obs})$. Continuous query $q \in \mathbf{Q}$, where $\mathbf{Q}$ is the set of all possible queries, is defined as a 5-tuple $q = (qid, x^{\ulcorner}, x^{\urcorner}, y^{\ulcorner}, y^{\urcorner})$, where $qid \in \mathbb{N}$ is an unique query identification value, and $x^{\ulcorner}, x^{\urcorner}, y^{\ulcorner}, y^{\urcorner}$ respectively are query rectangular window projections onto x- and y-axis of a 2-dimensional space.

If we define the set of all system tracked objects by $\mathbf{Objs} \subseteq \mathbf{O}$ and the set of all maintainable queries by **Queries** $\subseteq$ **Q**, then required system operations are defined as follows:

- **registerQuery**$(q \in \mathbf{Q}) =$
  $\langle \mathbf{Queries} = \mathbf{Queries} \cup \{q\}\rangle$

  Registers query q in the system.

- **dropQuery**$(qid \in \mathbb{N}) =$
  $\langle \mathbf{Queries} = \mathbf{Queries} \setminus$
  $\{q | q \in \mathbf{Queries} \wedge q.qid = qid\}\rangle$

  Removes query with id equal to *qid* from the system.

- **reportObjLocation**$(o \in \mathbf{O}) =$
  $\langle \mathbf{Objs} = \mathbf{Objs} \setminus$
  $\{d | d \in \mathbf{Objs} \wedge d.oid = o.oid\} \cup \{o\}\rangle$

  Inserts or updates location data for object with id equal to *o.oid*.

- **dropObj**$(oid \in \mathbb{N}) =$
  $\langle \mathbf{Objs} = \mathbf{Objs} \setminus$
  $\{o | o \in \mathbf{Objs} \wedge o.oid = oid\}\rangle$

  Removes object with id equal to *oid* from the system.

- **executeQuery**$(qid \in \mathbb{N}, t^{\ulcorner}, t^{\urcorner}) =$
  $\{o | o \in \mathbf{Objs} \wedge$
  $\exists t \exists q (q \in \mathbf{Queries} \wedge q.qid = qid \wedge$
  $(t_{now} \leq t^{\ulcorner} \leq t \leq t^{\urcorner} \leq t_{now} + d_{max}) \wedge$
  $(q.x^{\ulcorner} \leq x(\vec{loc}(o,t)) \leq q.x^{\urcorner}) \wedge$
  $(q.y^{\ulcorner} \leq y(\vec{loc}(o,t)) \leq q.y^{\urcorner}))\}$

  Retrieves a set of objects which intersect the rectangle of query with id equal to *qid* in the time window $(t^{\ulcorner}, t^{\urcorner})$ and where x and y are corresponding vector components, t_{now} is a current time, d_{max} is a largest possible time window duration, $\vec{loc}(o,t)$ is the previously defined object location prediction function.

4 The Proposed Solution

A straightforward brute force solution that provides the functionality, described in Section 3, would be an implementation of a basic application that uses main or disc memory to store all registered queries and latest movement behaviour data for each object. Then **executeQuery** operation result would be constructed by sequential looping through all collected movement data of objects in the set **Objs** and checking if object intersects the query's rectangle within a given temporal constraint. This solution does not provide optimal number of analysed objects per query evaluation, because all objects being outside the query spatial scope are analysed, which is inefficient in most cases. Spatial index on these moving objects can increase query evaluation performance, however when the number of tracked objects increases, the frequent index updates become very costly, as mentioned in Section 2.

We propose a more efficient technique for the construction of **executeQuery** result by exploiting the assumption that spatial constraints of every query are static and can be indexed. This index can be utilised to provide rapid identification of queries that will be intersected by any moving object from current time to infinity. If we perform such query identification for all moving objects and apply the location prediction function to calculate time moments when the query spatial constraint is satisfied, then we have enough data to construct the **executeQuery** result for any query in **Queries**. An **executeQuery** result for some query with id equal to *qid* can be constructed by collecting all objects, which satisfy query's spatial constraint during the time interval $(t^{\ulcorner}, t^{\urcorner})$. In comparison to brute force method, this solution provides lower number of analysed objects per query evaluation, thus all objects being outside spatial scope of the query will not be analysed. Our solution also presents a practical technique for construction and maintenance of the query-intersected object collection, that is utilised to provide results for **executeQuery**.

We utilise a spatially enabled relational database (RDBMS) together with spatial data indexing and spatial query evaluation features, to provide an environment for the implementation of our solution. It will be described in terms of a conceptual data model and operations.

Figure 1 shows an ER-diagram that defines the proposed data model. The entities "Moving Object" and "Query" correspond to the sets **Objs** and **Queries**. The "Intersecting" entity is introduced in order to relate intersected queries and moving objects and to provide a repository for intersecting time moments of query spatial constraint for each intersected object-query pair. Due to the assumption that query spatial constraint is described by a rectangle, only two attributes, "timeIncludes" and "timeExcludes", are used to define predicted absolute time moments of object trajectory and query rectangle intersection.

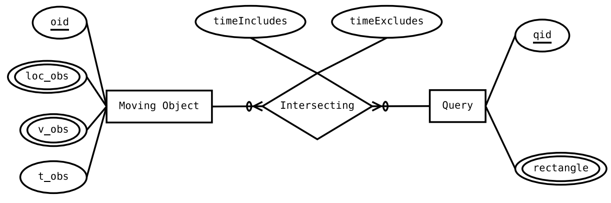

Figure 1: ER-diagram describing the system entities and their relations

Our conceptual data model is converted into a database schema with the corresponding tables *Objs*, *Queries*, and *Intersection* on some concrete RDBMS. To speed up data accessing in later defined operations, our solution requires following indices:

- Index for primary key attribute *oid* in *Objs*.
- Index for primary key attribute *qid* in *Queries*.
- Indices for attributes *oid* and *qid* in *Intersection* in order to speed up row selection for given *oid* or *qid*.
- Spatial index (using R-tree, Quad-tree, etc.) for the *rectangle* attribute in *Queries* in order to provide performance for the "object trajectory line and query rectangle intersection" SQL queries.

The operations **dropQuery** and **dropObj** can be implemented using basic SQL statements, however the execution of **dropQuery** requires updating spatial index. The performance of this operation mainly depends on the employed index type and is costly in most cases.

In our proposed solution, the **reportObjLocation** function is extended to provide object-query intersection computations. Each time this function with latest location data of any object o is executed, the table *Intersection* is updated with exact intersection time moments between $o.t_{obs}$ and a time moment close to infinity for each intersected pair (o, q), $q \in Queries$. Here we present a more general function **reportObjLocationBound**$(o \in \mathbf{O}, t^{\ulcorner}, t^{\urcorner} \in \mathbb{R})$, which can be easily adopted to compute **reportObjLocation**. It calculates the predicted trajectory of object o and each query rectangle's intersection time moments that fall only into the bounded time interval $[t^{\ulcorner}, t^{\urcorner}]$, $o.t_{obs} \leq t^{\ulcorner} \leq t^{\urcorner} < \infty$. It is defined in Algorithm 1, where notations are used as follows:

IsLRinters(l, r) - returns *true* for spatially intersected pairs of line segment and rectangle (l, r), where $l = (\vec{p_1}, \vec{p_2})$ is defined as a 2-tuple of line segment start and end points, rectangle $r = (x^{\ulcorner}, x^{\urcorner}, y^{\ulcorner}, y^{\urcorner})$ is specified by its projections $(x^{\ulcorner}, x^{\urcorner})$, and $(y^{\ulcorner}, y^{\urcorner})$ onto x- and y-axis of 2-dimensional space respectively.

LRinters(l, r) - returns a line segment, defined as $(\vec{p_1}, \vec{p_2})$, describing the intersection of each pair (l, r), where IsLRinters(l,r) holds.

```
   Input: $o \in \mathbf{O}, o = (oid, \vec{loc_{obs}}, \vec{v_{obs}}, t_{obs}); t^{\ulcorner}, t^{\urcorner} \in \mathbb{R}$

 1 delete from $Intersection$ where oid=$o.oid$;
 2 delete from $Objs$ where oid=$o.oid$;
 3 insert into $Objs$ values $(o)$;
 4 $\mathbf{l} \leftarrow (o.\vec{loc_{obs}} + o.\vec{v_{obs}} \cdot (t^{\ulcorner} - o.t_{obs}), o.\vec{loc_{obs}} + o.\vec{v_{obs}} \cdot (t^{\urcorner} - o.t_{obs}))$;
 5 select * from $Queries$ into $IQ$ where IsLRinters($\mathbf{l}$, $rectangle$);
 6 foreach $q \in IQ$ do
 7     if $(|o.\vec{v_{obs}}| = 0)$ then
 8         insert into $Intersection$ values $(o.oid, q.qid, t^{\ulcorner}, t^{\urcorner})$;
 9     else
10         $\mathbf{l_{int}} \leftarrow$ LRinters($\mathbf{l}, q.rectangle$);
11         if $|x(o.\vec{v_{obs}})| \geq |y(o.\vec{v_{obs}})|$ then
12             $t_1 \leftarrow t_{obs} + \frac{(x(\mathbf{l_{int}}.\vec{p_1}) - x(o.\vec{loc_{obs}}))}{x(o.\vec{v_{obs}})}$;
13             $t_2 \leftarrow t_{obs} + \frac{(x(\mathbf{l_{int}}.\vec{p_2}) - x(o.\vec{loc_{obs}}))}{x(o.\vec{v_{obs}})}$;
14         else
15             $t_1 \leftarrow t_{obs} + \frac{(y(\mathbf{l_{int}}.\vec{p_1}) - y(o.\vec{loc_{obs}}))}{y(o.\vec{v_{obs}})}$;
16             $t_2 \leftarrow t_{obs} + \frac{(y(\mathbf{l_{int}}.\vec{p_2}) - y(o.\vec{loc_{obs}}))}{y(o.\vec{v_{obs}})}$;
17         end
18         insert into $Intersection$ values $(o.oid, q.qid, min(t_1, t_2), max(t_1, t_2))$;
19     end
20 end
```

Algorithm 1: **reportObjLocationBound**

The time complexity of Algorithm 1 is linear with respect to the number of queries in *Queries*. Its performance would be increased and I/O operations count would be reduced by utilising spatial index that is build on the *rectangle* attribute in the *Queries* table for operation evaluation at line 5 in Algorithm 1. For most registered query datasets the index can be used efficiently to eliminate false object trajectory and query rectangle intersections. However it is not possible for all dataset types to guarantee a worst-case performance, that is better than the case, where index is not used [3].

The **reportObjLocationBound**$(o, t^{\ulcorner}, t^{\urcorner})$ function with parameter settings $t^{\ulcorner} = o.t_{obs}$ and $t^{\urcorner} = \infty$ computes exactly the same result as **reportObjLocation**(o). However the latter function may not be efficient enough for some real-time applications, where object movement parameters expire and are normally updated after a finite time duration. This function searches unnecessarily for all intersected queries into infinity assuming that the objects will move with fixed speed and direction. A possible improvement is to consider a database solution setting, where it is enough to provide result for **executeQuery** parametrised for a finite d_{max} time units into the future. If we only require an **executeQuery** result for a limited time window, a **reportObjLocation** call for some object can be substituted with periodical **reportObjLocationBound** calls providing shifted time windows as time progresses. A database solution, that uses this approach, will be presented in the next section.

The operation **executeQuery** is defined in SQL in Algorithm 2. To improve query evaluation performance, the created index for the *qid* attribute in the *Intersection* table can be applied.

As presented in Section 3, the definitions of **reportObjLocation** and **executeQuery** allow to distinguish two possible versions of **registerQuery**. The first version gives immediate correct **executeQuery** result after the query has been registered by **registerQuery**,

```
Input: $qid \in \mathbb{N}; t^{\ulcorner}, t^{\urcorner} \in \mathbb{R}$

1 select oid from $Intersection$ where
  $Intersection$.qid = $qid$ and
  (timeIncludes between $t^{\ulcorner}$ and $t^{\urcorner}$) or
  (timeExcludes between $t^{\ulcorner}$ and $t^{\urcorner}$) or
  ($t^{\ulcorner}$ between timeIncludes and
  timeExcludes)
```

Algorithm 2: **executeQuery** function

while the second version provides only incomplete results until all object locations are updated by **reportObjLocation**. The latter version of **registerQuery** just inserts a new record in the *Queries* table on each function call. Since the *Intersection* table is not updated, **executeQuery** can return an incomplete set of query-intersected objects until all the objects in the query's scope update their locations by executing **reportObjLocation**.

This issue is eliminated by using the first version of **registerQuery**. For each **registerQuery** call, in addition to the previously described *Queries* table update, all objects that intersect with the new query are found and concrete query rectangle intersection timings are computed and stored in the *Intersection* table. This will guarantee that a complete **executeQuery** results for this query will be available immediately after execution of **registerQuery**. Our solution avoids using index on moving objects due to performance reasons that were introduced in related work section. Consequently the query-intersected objects can not be selected efficiently and a costly linear scan of the *Objs* table must be applied. The possibility to tolerate incomplete results versus an expensive operation has to be considered before choosing either the first or the second version of **registerQuery** for a concrete system implementation.

A summary of the proposed solution is, that each time object movement behaviour is observed and delivered to the system, result sets of intersected continuous queries are updated immediately. We expect that this operation will be carried out efficiently with the help of the existing index on continuous queries. An evaluation of any of these queries only concerns the precomputed result selection from a table, followed by straightforward filtering, in order to select moving objects for a given time window. This approach provides efficient continuous spatio-temporal query evaluation, the performance of the location update operation is uncertain. We expect that it is still acceptable and that our solution is suitable for practical applications in a setting, where number of moving objects is higher than number of queries. This will be addressed in Section 6, where performance results of a prototype system will be presented.

5 Object-query Intersections Update Strategies

If we are only interested in continuous query result for a bounded (by d_{max}) time window, an optimisation can be performed in order to avoid the costly **reportObjLocation** execution. Concrete optimisation strategies mainly depend on the application where this system is being used. Next, we identify some of these applications and possible continuous query result precomputation strategies (i.e., result set update strategies).

5.1 Single Update Strategy

In an application, where we require that objects have to report their location once per fixed time duration o_{upd} in order to consider them alive, it is enough to execute function **reportObjLocationBound** once for $[o.t_{obs}, o.t_{obs} + o_{upd} + d_{max}]$ time window when a new location is delivered to the server. When current time progressed and becomes larger than $o.t_{obs} + o_{upd}$, then the object, if it has not delivered its new location to the server, will not be available for queries with time window as large as d_{max} and will disappear totally from the **executeQuery** result set when current time becomes $o.t_{obs} + o_{upd} + d_{max}$. This situation is depicted in Figure 2. The location of object o_1 is observed at t_{obs} and its trajectory and query intersection results are precomputed for $o_{upd} + d_{max}$ time units by executing **reportObjLocationBound**$(o_1, o_1.t_{obs}, o_1.t_{obs} + o_{upd} + d_{max})$. The precomputed intersection results can be fully utilised to evaluate the **executeQuery** result for any time windows in range $[t_{cur1},\ t_{cur1} + d_{max}]$, however precomputed results become insufficient for time windows in the range $[t_{cur2},\ t_{cur2} + d_{max}]$ and is not available at all for $[t_{cur3},\ t_{cur3} + d_{max}]$ range. The latter two cases can be interpreted as follows: "An object did not report its location within the required time period, so let us consider the object dead (or disconnected from the system) and adapt **executeQuery** result sets accordingly".

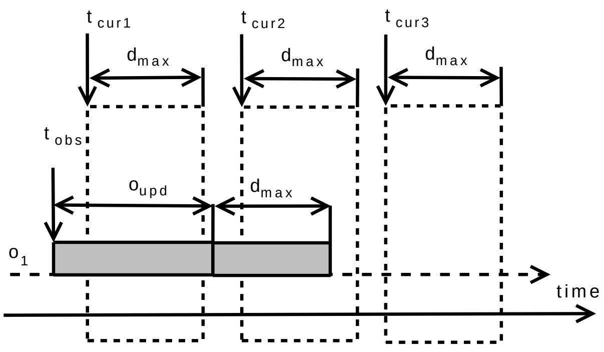

Figure 2: Single object with expiring locations querying diagram

5.2 Virtual Updates Strategy

In certain applications, it is impossible to specify o_{upd} values that are significantly small to provide the optimisation strategy, described in the previous section. An example of such application is tracking of semi-stationary objects, where each object rarely changes its location. In this case, we can not treat them as dead after some o_{upd} and have to include them into the continuous query result sets. Another example is tracking of very well predictable objects (like satellites, space shuttles, etc.) where forced periodical location updates, that are just issued to make sure the object is alive, are too costly. In these cases, a small o_{upd} value can not be defined. A possible **reportObjLocation** execution avoidance optimisation strategy for such applications, can be achieved using a so called virtual update policy.

As we introduced previously, execution of **reportObjLocation** can be substituted with a set of **reportObjLocationBound** calls. Each time a new location data of some object o is observed, instead of calling **reportObjLocation** we call **reportObjLocationBound** for a time window $[o.t_{obs}, o.t_{obs} + o_{vu} + d_{max}]$, where $o_{vu} > 0$. The system can then correctly include object o into the **executeQuery** result for o_{vu} time units from its location observation time $o.t_{obs}$. For the system to provide correct results of queries about object o for infinite time, **reportObjLocationBound** must be called repeatedly no later than d_{max} time units before the intersection data of o and queries expires. For finite or infinite repeated execution of **reportObjLocationBound** in order to keep query result sets updated, we define the term *virtual updates*.

We propose using $[t_{cur},\ t_{cur} + o_{vu} + d_{max}]$ time window, where t_{cur} denotes current time, for each **reportObjLocationBound** call in virtual updates. The reexecution of

reportObjLocationBound with this time window grants, that object o will be correctly included into the **executeQuery** result for additional o_{vu} time units into the future. A work diagram of the system, that applies virtual updates for some object o_1, is depicted in Figure 3. Due to virtual updates, the intersection data of o_1 is kept consistent for time interval $[t_{obs},\ t_4]$ and suitable for querying with temporal constraint in range between current time and d_{max} time units into the future. The actual intersection data recomputations are issued at time moments t_{obs}, t_1, t_2, and t_3, however no later than the deadlines, which respectively are $t_{obs} + o_{vu}, t_1 + o_{vu}$, and $t_2 + o_{vu}$.

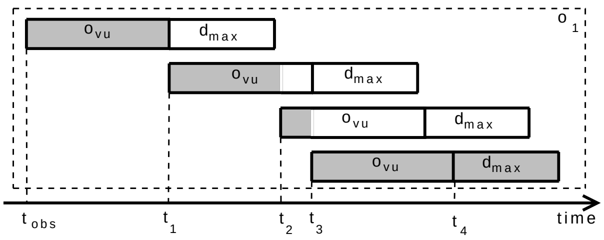

Figure 3: Single object virtual update diagram

5.2.1 Virtual Updates Scheduling

In order to efficiently apply virtual updates for some object o in a practical implementation of the system, each reexecution of **reportObjLocationBound** must be delayed as long as possible, but no later than the deadline, which is d_{max} time units before o's intersection data expires. This means that after o's new location data has been committed by the **reportObjLocationBound**(o, $o.t_{obs}$, $o.t_{obs} + o_{vu} + d_{max}$) call, the first following intersection data virtual update must be performed as close as possible to, but no later than the $o.t_{obs} + o_{vu}$ deadline. If **reportObjLocationBound** reexecutions are more frequent than $1/o_{vu}$, costly computational resources will be wasted to recompute already existing results. This overlapping of existing and new intersection data can not be avoided if we want to ensure a continuously correct **executeQuery** result in the system, however it could be minimised with the help of efficient scheduling algorithms. Such scheduling algorithms are the topic of real-time systems development research and will not be thoroughly discussed in this paper.

5.2.2 Virtual Updates Algorithm

One possible straightforward algorithm for scheduling, a modification of Earliest Deadline First, could be considered in a practical system implementation. The idea is to have a list of objects, where all entries are sorted in ascending order according to their next virtual update deadline. While time progresses, the entry with the earliest deadline is taken from beginning of the list. The entry's object intersection data is updated and a new entry with this object and its new virtual update deadline is inserted at the end of the list. To avoid overloading the system with unnecessary recomputations, the virtual update for the object in the entry at the beginning of the list, should be postponed if its deadline is relatively far in the future. In cases, where real location data of some object has been committed to the system and its intersection data has been updated, the existing object's entry in the virtual update list is always moved to the end with the new deadline.

One way to support virtual updates in our proposed RDBMS solution, described in Section 4, is to add a new deadline attribute to the *Objs* table and build an index on it.

6 The Prototype

In this section we present performance study results of our solution, integrated in a prototype system, that corresponds to a realistic case with a database server, moving objects and query subscribers. To be able to approximately evaluate capabilities of a real system, we design the following prototype stress tests on the most time consuming, throughput-limiting tasks in the system:

Test 1: A multi-object location update stress test. The aim is to measure a maximum number of system handled object location updates per second (location update frequency), emulating possible movement and location reporting behaviours of real objects and using different number of registered queries and lengths of intersection computation time window ($o_{upd} + d_{max}$ values).

Test 2: Infinite prediction, single and virtual update strategy comparison test. The aim is to measure the execution time, needed to process a fixed workload of moving objects in a prototype system using different query result update strategies (infinite prediction, single and virtual update).

Test 3: Virtual update strategy performance test. The goal is to approximate the optimal time window ($o_{vu} + d_{max}$ value), using a fixed amount of queries and workload.

6.1 Design of the Prototype

To accomplish the previously defined tests, our prototype is adapted to emulate a simple location based service domain, where artificial, GPS enabled, and on-line cars are tracked and queried. To perform prototype stress tests using a variable number of cars with customizable movement patterns, a special "Network-based Generator of Moving Objects [1]" tool is used. The generator simulates the natural behaviour of cars in a given street network and allows exporting the location data to some external application. The architecture of the prototype, that works with synthetic data to emulate previously defined domain, is shown in Figure 4. The *object client* emulator delivers simulated object locations from the automatically generated dataset to the RDBMS through a *service* emulator. This *service* emulator is used only to expose a RDBMS communication interface to *object clients* and *query subscribers.* A *query subscriber* emulator registers and invokes continuous spatio-temporal queries in the RDBMS.

The prototype is created and tested with the following settings:

- Simulated cars movement patterns follow San Francisco Bay area road network.
- Database server runs on a Intel Xeon dual core platform, with 2Gb RAM running Windows XP.
- Database server uses Oracle 10g RDBMS with spatial support.
- Operations **reportObjLocationBound**, **registerQuery**, **executeQuery** and others are implemented as PL/SQL stored procedures in the RDBMS to be executed internally in the server.
- Random sized queries are generated within the bounds of the simulation area.
- Query rectangles are indexed using the R-tree spatial index.
- Emulator software is implemented in C#.
- Virtual update deadlines, that are employed by the virtual update strategy, are maintained and stored in the emulator.

In most applications the virtual updates deadlines will be stored in the RDBMS, as mentioned in Section 5. This can give a performance penalty for maintaining the index on the deadline attribute. We do not consider this penalty to be significant.

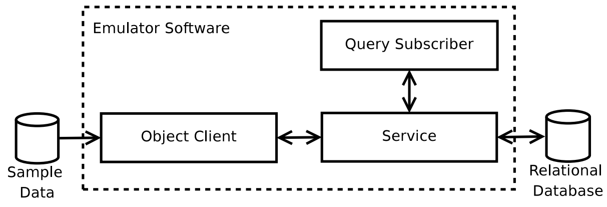

Figure 4: Architecture of the prototype system

6.2 Design of Test Cases

For all tests, a single workload is generated. It contains 2000 records, where each defines location, direction, and speed of some object in the area at a specific time. The workload contains simulated location data for 250 moving objects. In progress of the simulation, the number of objects grows from 50 to 250. At any given time in the simulation, there exist objects that no longer move or do not report their location. In all tests, the workload is fed to the prototype applying the shared prediction technique. As a result, only 1572 of the 2000 records are actually submitted to the RDBMS.

We describe the prototype setting used to carry out our tests as follows:

Test 1 computes the intersection data using the **reportObjLocationBound** function with time window length $(o_{upd} + d_{max})$ values equal to 30, 60, 90, and a close to infinity number of seconds. Maximum update frequencies in this experiment are computed from measured durations of the 1572 location records, submitted to the system, having 500 to 5000 queries registered.

Test 2 is used to compare infinite prediction, single, and virtual update strategies impact on the prototype system throughput. For each prototype system configuration smaller sub-tests are constructed with different update strategies and parameter settings. Each of the sub-tests measures the execution time, needed to run the previously mentioned workload on the system using a varying number of registered queries (values equalling to 100, 500 and 1000 are selected). A correspondence between each sub-test and system configuration is defined as follows:

- *Test 2.1*: Infinite prediction update strategy applied.
- *Test 2.2, 2.3, 2.4*: Single update strategy with the prediction time window lengths $(o_{upd} + d_{max})$ equalling to 20, 40, and 60 seconds applied.
- *Test 2.5, 2.6, 2.7*: Virtual update strategy with different prediction time window lengths $(o_{vu} + d_{max})$ equalling to 20, 40, and 60 seconds applied.

Test 3 measures the time needed to run the previous mentioned workload on the prototype, using the virtual update strategy and 500 registered queries, with a time window going from 20 increasing by 2, up to 80 seconds.

6.3 Prototype Test Results

Figure 5 shows results of *Test 1.* In this figure we see that, when queries randomly cover the whole simulation area, the intersection data computation for a 30 seconds time window provides almost two times better location submission performance in comparison to the case, where an infinite time window is used. The number of registered queries in the system does not have obvious impact on the ratio between the performance of the experiments with the infinite and the lowest time windows. The location update frequencies for different lengths of the time window become similar when the number of queries increases. This could be a consequence of the increased density of the random query coverage in every point of the simulation area, when even slowly moving objects intersect with a large number of queries per short time interval. The concrete frequency values are sufficient for most practical applications, considering that they can be controlled by changing server hardware specifications and coverage of the queries.

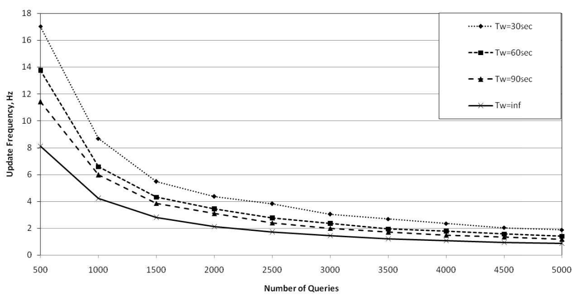

Figure 5: Results of *Test 1*

Grouped individual sub-test results of *Test 2* are presented in Figure 6. As expected, 1572 location submissions to the system with infinite prediction applied (*Test 2.1*) in most cases have a longer execution time than the other tasks. There is an exception that when the number of registered queries and the length of the virtual update time window is small (*Test 2.5*), due to virtual updates overhead, the whole dataset submission is more time consuming than the test with infinite updates.

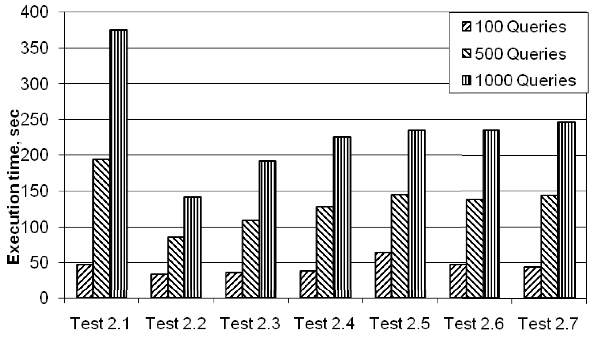

Figure 6: Results of *Test 2*

As expected, in *Test 2.2* the best system throughputs for all chosen query counts are observed. This is because there is no additional virtual updates interference and shortest time window is used for object-query intersection computations.

Tests 2.5, *2.6*, and *2.7* demonstrate that virtual updates efficiently simulate infinite prediction strategy. When the number of queries is high enough (in our experiment this is a case with 500 queries), virtual updates provide a better overall system performance in respect to the infinite update case.

Test 2 results show that the different object-query intersection update strategies are preferable in certain applications. E.g. the infinite prediction strategy would be useful in cases where long term object location prediction is required, while single updates could be the best choice for applications where the longest location update period o_{upd} is precisely defined for all objects. But usually the virtual update strategy would be preferred, when many object locations must be maintained for long or infinite time.

The result of *Test 3* is shown in Figure 7. It shows the execution time and the number of virtual updates for a given time window, when using the previously described submission task and with 500 queries registered in the system. This experiment shows that virtual updates time window can be tuned to provide a better system throughput. When the time window length is short, intersection data of each object, that does not report its location for some period, must be updated frequently. This negatively affect throughput of observed object location updates. Virtual updates also negatively affect observed object location updates when the time window becomes large. This is due to significant system efforts to make intersection predictions for a long time duration, even if virtual update frequency is relatively low.

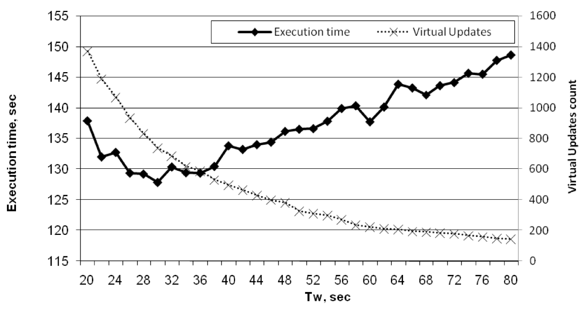

Figure 7: Results of *Test 3*

7 Conclusion

Motivated by an increased number of advanced location based services and velocity-vector-based location update policy, this paper proposes a general purpose database solution that, with support of efficient continuous queries on a large number of moving objects, can be used as a core element in most LBS systems.

The database solution is suitable for efficient evaluation of rectangular shaped continuous queries on current and predicted locations of moving objects in cases, where the number of objects is higher than the number of queries. The solution works with different types of moving or semi-stationary objects and supports object disconnection handling (handling of objects that did not reported their location after some period). We introduce a so called virtual update strategy to improve system performance in cases, where object locations are predicted into infinity.

Our solution is unique in that it combines spatial indexing on continuous queries and velocity-vector-based shared prediction. This combination enables efficient partial result precomputation of continuous queries on shared prediction enabled moving objects. Most related work utilises either query indexing without shared prediction support or object count limited solutions, that use spatial indexing on moving objects.

In this paper we present a prototype of our solution, implemented on a spatially enabled RDBMS, employing its features like spatial indexing and geometric intersection calculation. Our performance study shows that the prototype can accept a number of location updates, which is a bottleneck operation in the system, that would be sufficient for most practical applications. We show how different system configurations impact the prototype performance, and observe that each of the proposed object-query intersection update strategies could be preferable in certain applications.

Further tests could be performed on the implemented prototype, experimenting with different patterns for query rectangle arrangements. This would include patterns like clustering of queries and equally sized queries, for better emulation of different applications. In further research, it would be interesting to evaluate the performance of a new query registration, when immediate query results are required. Another modification of our solution that supports non-rectangular spatio-temporal queries would be interesting, emphasising that all required operations for this task are already present in most spatial enabled RDBMS. Other research could be carried out to identify cases where our solution performs better than a solution that uses spatial index on moving objects and vice versa.

8 Acknowledgement

We would like to thank our supervisor Simonas Saltenis for help and advise throughout the semester.

References

[1] Brinkhoff, T. A framework for generating network-based moving objects. *GeoInformatica 6*, 2 (June 2002), 153–180.

[2] Civilis, A., Jensen, C. S., and Pakalnis, S. Techniques for efficient road-network-based tracking of moving objects. *IEEE Trans. Knowl. Data Eng 17*, 5 (2005), 698–712.

[3] Guttman, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1984), ACM, pp. 47–57.

[4] ipoki.com. Ipoki - gps-based social network, share your location in real-time. `http://www.ipoki.com`, 2008.

[5] Kalashnikov, D., Prabhakar, S., Hambrusch, S., and Aref, W. Efficient evaluation of continuous range queries on moving objects. *Lecture Notes in Computer Science 2453* (2002), 731–740.

[6] Mokbel, M. F., Xiong, X., Aref, W. G., Hambrusch, S. E., Prabhakar, S., and Hammad, M. A. PLACE: A query processor for handling real-time spatio-temporal data streams. In *VLDB* (2004), M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, Eds., Morgan Kaufmann, pp. 1377–1380.

[7] Prabhakar, S., Xia, Y., Kalashnikov, D. V., Aref, W. G., and Hambrusch, S. E. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Computers 51*, 10 (2002), 1124–1140.

[8] Saltenis, S., Jensen, C. S., Leutenegger, S. T., and Lopez, M. A. Indexing the positions of continuously moving objects. *SIGMOD Record (ACM Special Interest Group on Management of Data) 29*, 2 (2000), 331–342.

Resume of: "Continuous Queries on Current and Predicted Locations of Moving Objects"

Paper by:
Laurynas Siksnys, Katja Springer, Simon Nicholas M. Tinggaard, Rune L. Wejdling
Resume by:
Simon Nicholas M. Tinggaard, Rune L. Wejdling

The papers main focus is on the fact that there has been an increase in the number of advanced location based services (LBS) over the recent years. The amount of location aware devices rises rapidly increasing the focus on this subject. LBS face a great challenge when the number of users becomes large, because the number of location updates submitted to such a system can become huge.

The paper utilises recent developed techniques, like velocity-vector-based shared prediction updates strategy, to lower the number of updates needed by the system, while maintaining a high level of location prediction accuracy. Another goal of a LBS is to give the users a quick response to any given request. This lays the setting for another technique utilised in the solution proposed in the paper. To speed up the query response time the proposed solution uses precomputation of spatial queries, in order to achieve this efficiently a spatial index is build on the queries registered in the system.

Most of these subjects have been discussed in related work, but the solution in the paper is unique due to the fact that it combines spatial indexing on continuous queries and velocity-vector-based shared prediction. Performance studies are provided with a prototype implementation, as a stored procedure, on a commercial Relational Database Management System (RDBMS), to show the effectiveness of this solution under certain workloads. A small program is implemented to emulate moving objects location submission behavior, by executing generated workloads on the prototype.

Section 4 of the paper provides the algorithms of the solution and describes how these can be used to compute the current and future location of moving objects. Every time an object submits a updated location to the system, the main algorithm is executed, computing the intersections between the object and the registered queries within the given time window.

In section 5 the paper introduces three different update strategies that are later implemented and compared in the prototype. These update strategies differ mainly on how long time into the future the system can return valid results. The first strategy simply calculates the objects trajectory and query intersections into infinity. This is shown in a performance study of the prototype, not to be a very good all round solution. The second strategy calculates the objects trajectory and query intersections only for some finite time into the future, which gives good performance in the prototype test. This strategy has the disadvantage that the moving objects have to submit location information more often in order to stay in the systems query results. The third presented update strategy introduces the term virtual updates, which keeps track of the moving objects last location submission and executes periodical updates to ensure a valid result.

Performance results of the prototype, shows that the virtual updates strategy would be a efficient solution in most general purpose cases, but the other strategies will perform better in some special cases.