Mobile Information Systems 5 (2009) 205–232
DOI 10.3233/MIS-2009-0082
IOS Press

Location-dependent query processing under soft real-time constraints

Zoubir Mammeri, Franck Morvan, Abdelkader Hameurlain* and Nadhem Marsit
IRIT, Paul Sabatier University, Toulouse, France

Abstract. In recent years, mobile devices and applications achieved an increasing development. In database field, this development required methods to consider new query types like location-dependent queries (i.e. the query results depend on the query issuer location). Although several researches addressed problems related to location-dependent query processing, a few works considered timing requirements that may be associated with queries (i.e., the query results must be delivered to mobile clients on time). The main objective of this paper is to propose a solution for location-dependent query processing under soft real-time constraints. Hence, we propose methods to take into account client location-dependency and to maximize the percentage of queries respecting their deadlines. We validate our proposal by implementing a prototype based on Oracle DBMS. Performance evaluation results show that the proposed solution optimizes the percentage of queries meeting their deadlines and the communication cost.

Keywords: Databases, mobility, location-dependent query processing, real-time constraints

1. Introduction

Mobile units allow users not only to query a database anytime and anywhere, but also to send queries the results of which are location-dependent. These queries are called *Location-Dependent Queries* (LD-queries). For instance, a doctor uses his/her Personal Digital Assistant and asks for the addresses of hospitals located within 15 km far from his/her current position. Many new services and applications need to effectively process these types of queries. This generated a great deal of interest from the scientific community. In fact, several researches examined various problems related to location-dependent query processing [8,27,31]. Issues such as expressing LD-queries or taking into account mobile client location while processing queries have been addressed in many works [8,9,18,23]. The representation and querying of moving objects (e.g. select the closest free ambulance) have also been considered by several works [5,16,24,27]. In spite of their abundance, the proposed solutions do not meet all the location-based application needs. In fact, the real-time constraints of some applications have not been considered enough. Such new requirements generate the need to consider new types of queries such as location-dependent queries with real-time constraints. These constraints often appear as deadlines associated with queries. For example, a doctor asks for the closest pharmacy and he/she requires receiving the query results within 30 seconds. The deadlines can be explicitly specified by users or derived from user mobility. Indeed, the location-dependent query results have time-limited validity which depends on the users' movement between sending the query and receiving the results. For example, the closest

*Corresponding author: Abdelkader Hameurlain, IRIT, Paul Sabatier University, 118 Route de Narbonne, 31062 Toulouse, France. Tel.: +33 5 61 55 82 48; Fax: +33 5 61 55 62 58; E-mail: hameurlain@irit.fr.

pharmacy may change when the doctor moves only by 500 meters (if doctor vehicle speed is 60 km/h the query result becomes incorrect after just 30 seconds). Even though several researches have been dedicated to developing Real-time DBMS (RTDBMS) [14,29,30], these DBMS do not take into account the LD-queries characteristics (e.g. client mobility, location-dependency of query results).

The objective of this paper is to propose an efficient solution for LD-query processing under real-time constraints. The architecture proposed in this paper is targeting soft real-time applications (i.e., applications which have explicit or implicit deadlines but in case of deadline missing there is no severe or dramatic consequences). First, we present a software architecture allowing processing of various types of LD-queries. The modules of our architecture are designed to be implemented on top of conventional DBMS (e.g. Oracle). Second, we focus on some modules and we propose a method to consider the query result location-dependency. In fact, we propose to transform LD-queries before submitting them to the underlying DBMS in order to bind the client location and possibly to take into account his/her direction. In order to reduce communication cost, we propose to reject (before the data transfer) the incorrect query results (due to client location change). Third, we focus on functions that provide some real-time capabilities. Therefore, methods are proposed in order to maximize the percentage of queries meeting their deadlines (success ratio). We discuss the sources of deadlines in our context (e.g. explicit deadlines specified by users or implicit deadlines derived from client mobility). Then, we propose to apply an admission test in order to early reject queries which have no chance to meet their deadlines. We also propose to schedule queries according to their deadlines. Finally, we validate our proposal by implementing the proposed modules on top of Oracle. Performance evaluation results show that the proposed solution optimizes the percentage of queries meeting their deadlines and the communication cost.

The remainder of the paper is organized as follows: in Section 2, we present the related work. In Section 3, we present our assumptions, types of queries of concern, and an overview of the proposed architecture. In Section 4, we present mechanisms for location-dependency constraints. In Section 5, we present mechanisms for handling timing constraints. In Section 6, we present validity check of results before sending results to mobile clients. In Section 7, we evaluate the performance of our architecture. Section 8 is the conclusion.

2. Related work

[7] is one of the first works that highlighted the need of proposing new solutions for processing LD-queries issued by mobile units. In [19], the authors described the main steps for LD-query processing. They explained that it is necessary to bind client location to the query before executing it. The authors also claimed that it is important to add a new operator called the Location Related operator (LR-Operator) to express proximity conditions (e.g. the closest target object) or orientation conditions (e.g. straight ahead). Other work extended the existing languages in order to express different types of LD-query [8,18]. Another issue of concern is the validity of LD-query results. Solution proposed in [31, 33] is to compute, during the query processing, a particular region called the Validity Region (VR) such as the LD-query results remain unchanged whatever the client location is. The client should not resubmit the same query while being in the VR. This solution has some drawbacks like the overhead introduced by computing VR and by transmitting information about VR to users through the wireless network [32]. Several efforts have also been made to meet the need of querying moving object databases (i.e. databases in which data represent moving objects such as ambulances or helicopters) [25,27]. Two major problems have been addressed: (i) modeling and querying continuously changing data like moving

object locations [5,16,21] and (ii) tracking and updating the location of moving objects, and managing the uncertainty on location due to the imprecision of localization technology and to the continuous movement of objects [25,28].

Despite their abundance, previous works on LD-queries did not focus on LD-query processing under real-time constraints. In fixed environments, several solutions, combining techniques issued from the database field and the real-time system field, have been proposed. Since the end of the 1980s, many researches focused on Real-Time DBMS (RTDBMS). Those works mainly emphasized the problem of transaction scheduling and concurrency control [14,15,29,30]. Several algorithms have been proposed and various simulators and prototypes have been developed [11]. However, RTDBMS are not available on the market and the standardization is not yet advanced enough; especially when compared to conventional DBMS. This drawback makes it difficult to use RTDBMS at the core of location-based applications. In addition, the issues relating to mobility of client and/or objects have not been considered in depth by proposed RTDBMS. Among the rare proposals for taking into account both real-time constraints and mobility constraints we mention the work presented in [2,6,10]. Nevertheless, these works did not consider the characteristics of certain LD-queries (e.g. the dependency of query results according to location and/or direction). They rather focused on the impacts of the mobile environment characteristics (frequent disconnection, low wireless bandwidth) on real-time query processing. The aim of our work is to focus on timing constraints and location-dependency of query results while processing queries.

3. Assumptions, query specification, and overview of the proposed architecture

3.1. Assumptions

The mobile environment under consideration is composed by mobile clients communicating with fixed server through wireless and wired networks. We assume that the client has limited computational resources. Thus, the query processing is on the server side. We assume that the server is uni-processor and multi-users. In this work, we do not consider the problems that may be caused by latency variation due to wireless networks and we suppose that clients have guarantees on wireless network bandwidth [20]. To represent moving object locations in the database, we choose the simplest model among those presented in the literature [24]: only the last location given by the positioning system is stored in the database and it is updated periodically (e.g. every minute). In this case, methods for querying data related to fixed objects and methods for querying data related to moving objects are the same. However, it is not possible to query the future position of a mobile object. The use of a more complex model as those described in [24,25,27] to estimate this position generates heavy steps of supplementary process. These steps yield an overhead which may result in missed deadlines in a real-time context. In addition, we assume that only soft firm deadlines are associated with queries (i.e. if a deadline is missed, the query results are useless and they are rejected without dramatic consequences or damages).

Our objective is to provide an effective solution for taking into account real-time constraints while processing location-dependent queries. To reach this goal it is necessary to consider two types of constraints: (i) constraints related to mobility and query results location-dependency and (ii) real-time constraints. Indeed, it is necessary to propose methods for considering client location, proximity conditions (e.g. objects located within 10 km far from my current position) and orientation conditions (e.g. objects in front of me). In addition, due to client movement after sending the query, a part of the query results may become incorrect. So it is also necessary to avoid transmitting incorrect results to mobile clients. Since deadlines may be associated with LD-queries, it is also important to maximize the percentage of queries meeting their deadlines. A solution taking into account all these constraints needs to include techniques from different fields (databases, localization systems, real-time systems, wireless networks) that cohabit in a flexible and adaptable architecture.

3.2. Query specification and attributes

3.2.1. Forms of query specification

The types of client queries, which the architecture proposed in this paper is able to process, are classified according to their requirements and characteristics [12]:

- *Object mobility*: clients may search data related to fixed objects (e.g. hospitals) and or data related to moving objects (e.g. ambulances).
- *Location-dependency of query-result:* query results may depend on initial location (i.e. the client searches object relevant for his/her present location when sending the query), dynamically changing location (i.e. the mobile client asks for target objects close to his/her position at a given future instant) or future target[1] client location (i.e. the mobile client asks for target objects he/she can reach in less than certain time or certain distance). These queries are considered as spatiotemporal queries.
- *Query deadline:* when the processing of a given query should be completed before deadline, we talk about LD-queries with real-time constraints.
- *Result completeness:* in many situations users may accept partial results delivered as soon as possible rather than waiting a long-time for the complete but late results (e.g. a fireman searching for fire hydrants 15 km far from his/her position needs the results in less than 2 minutes and requires at least 2 fire hydrants).
- *Continuous queries:* this type of queries allows the clients “periodically” getting possibly changing results from a database without having to issue the queries repeatedly [3].

Thus, four main types of requirements may be specified in query clients:

- moving requirements,
- location-dependence requirements,
- timing requirements,
- result completeness requirements.

Moving requirements: since we are interested in mobile clients, the specification of client moving model is of prime importance. We propose the client query includes a moving profile which enables the calculation of the area (zone) relevant for searching objects and also for the deadline to send the results in such a way that the client receives the right objects at the right location. In the current version of our architecture, client moving profile must include speed and direction (south, north. . .). Optionally it may include the initial and final location coordinates. In the future, more realistic and powerful profiles should be included such as traffic conditions (to adjust the estimation of distance covered by mobile client), cartographic information (to take into account the real position of current and future location on a road map). . .

Location-dependence requirements: they involve proximity conditions, orientation conditions, and future client location conditions:

[1]Future target location is either the final client destination or some location relevant for client moving (e.g. a city, highway exit. . .) to make a decision.

Table 1
Examples of LDQ query specification using LDQ language

Location-dependent query examples	Queries expressed in LDQ language

Q1: select hotels names and addresses within 15 km far from my position	SELECT name, address
	FROM hotels WHERE AT DISTANCE (15 km)

Q2 : select names and addresses of the three closest hospitals	SELECT name, address
	FROM Hospitals WHERE CLOSEST NEIGHBORS (3)

Q3: select names and addresses of hotels reachable within 10 minutes driving South at 80 km/h	SELECT name, address
	FROM hotels WHERE MOVING PROFILE (80 km/h, DIRECTION SOUTH)
	TARGET LOCATION REACHABLE WITHIN (10 min)

Q4: select gas station addresses on my right along the next 50 km in my direction driving North at 110 km/h	SELECT address
	FROM GasStations WHERE MOVING PROFILE (110 km/h, DIRECTION NORTH)
	AT DISTANCE(50 km) ORIENTATION EAST

Q5: select addresses of gas stations 5 km from my position. I am driving in the North direction at 100 km/h.	SELECT address
	FROM GasStations WHERE MOVING PROFILE (100 km/h, DIRECTION NORTH)
	AT DISTANCE(5 km)

Q6: select hydrants 4 km far from my position. The results are required within 30 seconds.	SELECT address
	FROM Hydrants WHERE AT DISTANCE(4 km)
	TEMPORAL CONSTRAINTS (DEADLINE 30 s)

Q7: select addresses of gas stations 4 km far from my position during the next 30 minutes. An update is required every 2 minutes. I am moving to the West at 100 km/h.	SELECT address
	FROM GasStations WHERE MOVING PROFILE (100k/h, DIRECTION WEST)
	AT DISTANCE (4 km)
	TEMPORAL CONSTRAINTS (EVERY 2 min DURING 30 min)

Q8: select addresses of hospitals 10 km far from my position, at least 20% of the complete results is required within 60 seconds	SELECT address
	FROM hospitals WHERE AT DISTANCE (10 km)
	TEMPORAL CONSTRAINTS (DEADLINE 60 s)
	PARTIAL RESULT AT LEAST (20%)

– *Proximity* conditions can be expressed in four manners:
  - ∗ Searching for target objects such as the distance between their locations and the client position is less than a given distance (use of AT DISTANCE clause). E.g. query Q1 – Table 1.
  - ∗ Searching some nearest neighbors among target objects (using CLOSEST NEIGHBORS clause). E.g. query Q2 – Table 1.
  - ∗ Searching for the target objects that the client can reach within some access time at a given speed or reachable within a given distance (using of REACHABLE WITHIN clause). E.g. query Q3 – Table 1.
  - ∗ Searching for the target objects at a given distance from a fix location (using AT LOCATION clause).
– *Orientation* conditions can be specified (using ORIENTATION clause) to indicate where the objects should be located with regard to the client location. E.g. query Q4 – Table 1.

Timing constraints may be specified:

– either to fix a deadline for the results to be sent to the client (using DEADLINE clause). E.g. query Q6 – Table 1,
– or to specify that the results should be periodically updated by the server without resubmitting the query. E.g. query Q7 – Table 1.

Result completeness requirements: client may be interested only in a certain fraction (or threshold) of the objects that may be returned by the server. E.g. query Q8 – Table 1.

Below is the syntax of the language we propose to formally specify the queries. Table 1 provides some query examples to better understand how client queries are specified.

```
<query> ::= SELECT <set_of_target_object_attributes>
   FROM <set_of_target_objects>
   WHERE
   [ <client_moving_profile> ]
   <location_dependence_requirements>
   [ TEMPORAL CONSTRAINTS "(" <timing_requirements> ")" ]
   [ PARTIAL RESULTS <partial_results_requirements> ]
   [ ORDERED BY (<object_attribute> | DISTANCE TO "("<target_object_name> ")")]
   [ (AND | OR) <Not_location_dependent_clause> ]
<client_moving_profile> ::= MOVING PROFILE "(" <speed> <speed_unit> ","
   DIRECTION <direction> [ "," [ <initial_location> ] "," [ <final_location> ] ] ")"
<location_dependence_ requirements> ::= [ ORIENTATION <direction> ] <proximity_cond>
<proximity_cond> ::=
      AT DISTANCE "(" <distance><distance_unit> ")"
      | CLOSEST NEIGHBORS "(" <number_neighbors> ")"
      | [(BEFORE | CROSSING)] TARGET LOCATION
         (REACHABLE WITHIN "(" ( <time><time_unit> | < distance><distance_unit>)")"
         | AT <fixed_coordinates>)
<direction> ::= ( NORTH | SOUTH | WEST | EAST | NORTH EAST | NORTH WEST |
      SOUTH EAST | SOUTH WEST)
<timing_requirements> ::= <deadline_constraint> | <continuous_query_constraint>
   <deadline_constraint> ::= DEADLINE <expiration_time> <time_unit>
   <continuous_query_constraint> ::= EVERY <update_period> <time_unit>
      (DURING <duration><time_unit> | FOR <distance>)
<partial_results_requirements> ::= AT LEAST "(" ( <number> | <percentage>"%") ")"
   | AT MOST "(" <number> ")"
```

3.2.2. *Query attributes*

When a query is received, it is analyzed by Attributes and constraints identification function to determine the query type and its requirements (location, temporal, and result completeness requirements).

The management of queries during their life cycle (from the time they are received by the server till the transmission either of results or failure message to the clients) requires initializing and updating different parameters, which we call "query attributes". There are attributes for each kind of requirement.

Table 2 shows for each attribute the keyword which must be included in the query to determine the attribute value. For some specific attributes, if their initial value cannot be derived from the query, a default value is used. E.g. if the query does not include an AT DISTANCE clause (which defines the radius of the search area), a default value is used (e.g. 2 km for clients moving in a city and 25 km for clients on highway). Table 2 also shows other attributes of which values are either derived according to some rules explained in the rest of the paper.

To easy the manipulation of attributes and to shorten the notation, we adopt the following: for each attribute the values of this attribute for all the queries under management at the server are handled as a vector where the index is the query identifier (Q denotes the Id of the query under consideration). E.g. *Speed*(*Q1*) denotes the speed of the client of query $Q1$ and *Tserv*(*Q2*) denotes the estimated DBMS processing time of query $Q2$.

Table 2
Query attributes

Attribute category	Attribute name	Attribute description	Attribute specified in clause or calculated
Location-oriented			
	Speed	Client speed	MOVING PROFILE
	Dir	Moving direction of the client	MOVING PROFILE
	Orient	Orientation of the objects with regard to client location	ORIENTATION
	Radius	Max distance of objects from the client location	AT DISTANCE
	DistToGo	Distance between initial location and final location	REACHABLE WITHIN, AT
	$L0$	Initial client location	MOVING PROFILE
	Lf	Final client location (also called Future target location)	MOVING PROFILE, REACHABLE WITHIN, AT
	$L*$	Last client location estimate	
Timing-oriented			
	$t0$	Absolute time of query transmission by client	
	tqr	Absolute time of query reception by server	$tqr = t0 + Tqt$
	RelDeadline	Explicit Relative deadline for query processing	DEADLINE
	Duration	Time interval for continuous query evaluation	DURING, FOR
	Period	Period for continuous query evaluation	EVERY
	TimeToGo	Distance from initial to final location	REACHABLE WITHIN
	Tqt	Estimate of query transfer time	
	Tserv	Estimate of DBMS processing time	
	Trt	Estimate of result transfer time	
	Cost	Query response time	$Cost = Tqt + Tserv + Trt$
	AbsDeadline	Absolute deadline (used to EDF-scheduling)	
Others			
	CId	Identifier of the client submitting query. This Id is used by service location to provide client location.	
	LoZ	Indicates if client requires results before entering the zone around *Lf* or while crossing this zone	BEFORE, CROSSING
	PoolRes	Pool (buffer) storing continuous-query results	
	PartResMin	Minimum amount of results for partial result query	AT LEAST
	PartResMax	Maximum number of results for partial result query	AT MOST
	SpatialQuery	Spatial query yielded by LD query rewriting	

It should be noticed that all the attribute vectors are common variables shared by all the modules of the architecture.

3.3. *Overview of the proposed architecture*

Figure 1 summarizes the main steps of LD-query treatment from the reception of a query till the transmission of the corresponding results (each step is realized by a function included in our architecture):

- Carry out syntactic analysis of query to yield the values of the different parameters/requirements specified in the query and initialize query attributes shown in Table 1.
- Calculate the search area where the objects to return to client should be located.
- Rewrite the LD-query to yield an equivalent spatial query to be processed by the underlying DBMS.
- Submit the spatial query to evaluate its DBMS-related cost and then estimate the DBMS processing time and result transfer delay.
- Calculate the absolute delay to assign to query in order to meet client deadline, if any, and client movement and future location.

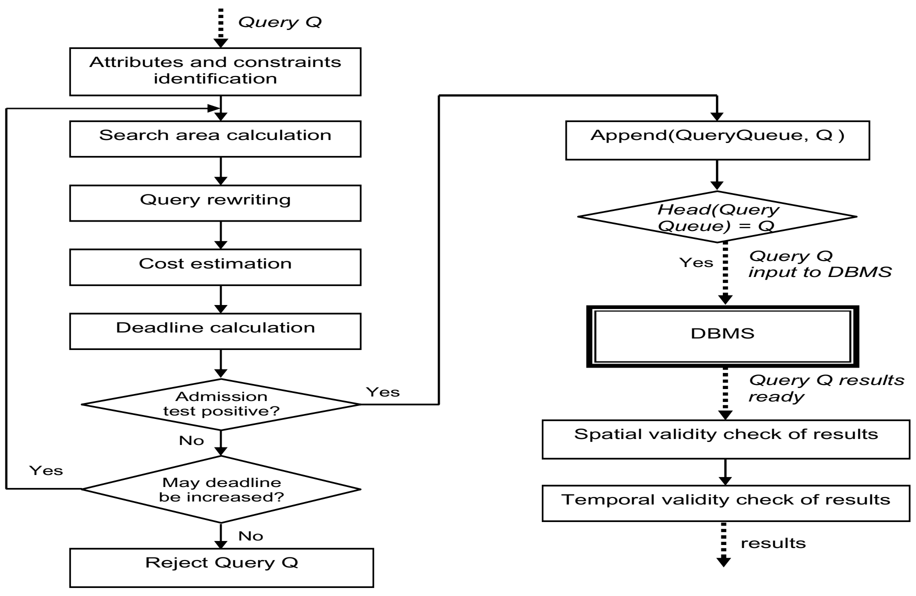


Fig. 1. Main steps of LD-query handling.

– Test if the timing constraints of the query may be met under the current load of the server. If the test is positive, the spatial query is admitted and inserted -according to its deadline- in the Ready-query queue. Otherwise, relax some constraints (change the implicit deadline or search area) and try if the query may be admitted. When all the attempts are unsuccessful the query is rejected.
– Process the query (when it is at the head of the ready query queue) and produce results.
– Check if the results are valid with regard to the current location of the client.
– Check if the results are still valid with regard to timing constraints and they may be transmitted on time.
– Transmit the valid results to the client.

Figure 2 illustrates the architecture components involved in location-dependent query treatment. There are five modules:

– *Query analysis and execution parameter calculation* module which includes four functions: Attributes and constraints identification, Search area calculation, Query rewriting, and Cost estimate.
– *Timing constraint management* module, which includes four functions: Deadline calculation, Admission test, EDF[2] query scheduler, and Continuous query manager (which is in charge of requesting the periodic processing of continuous queries).
– *Result check and management* module, which includes four functions: Spatial validity check of results, Temporal validity check of results, Continuous query result management (which is in charge of periodically updating the results sent to client while moving), and Query partial-result management (which selects among results produced by the DBMS a subset to be sent as soon as possible to the client).
– *DBMS* (with spatial capabilities) module, which processes queries and delivers results.

[2]EDF: Earliest Deadline First.

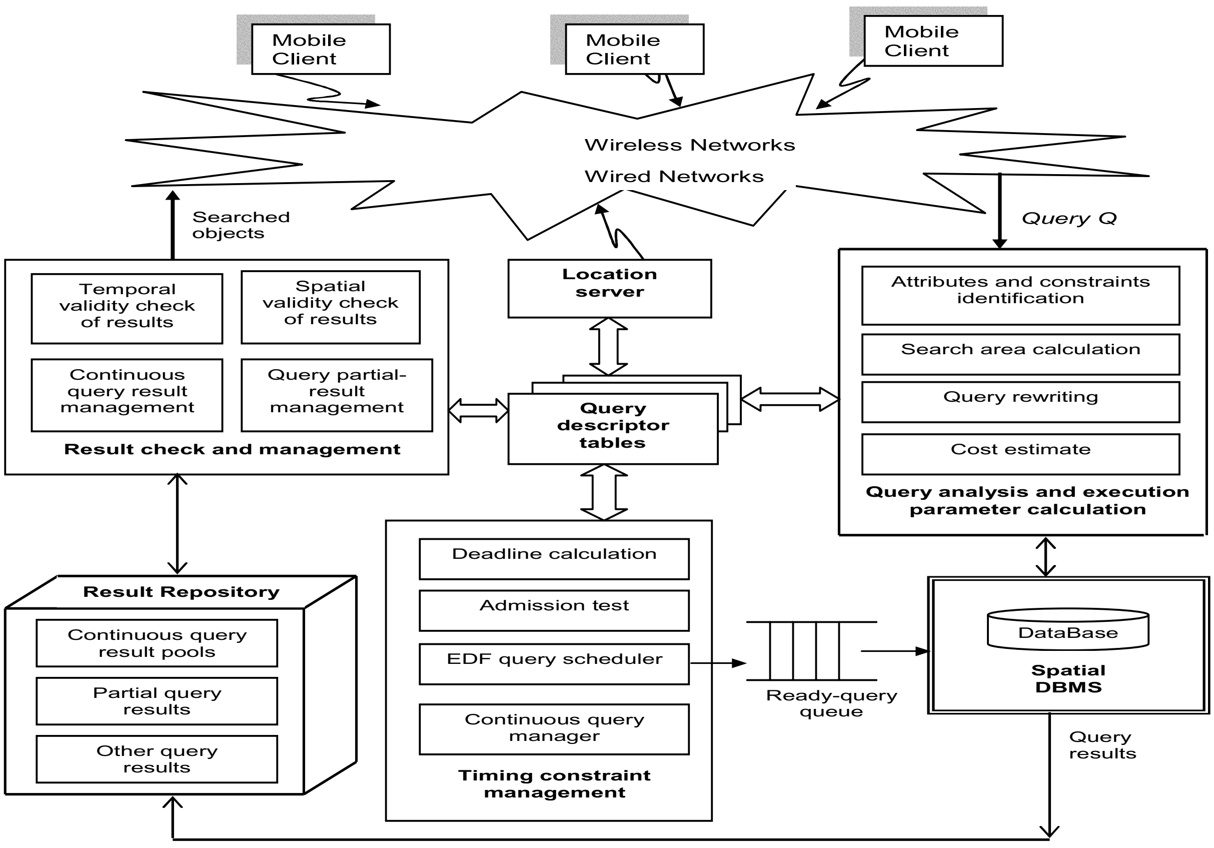


Fig. 2. Architecture for real-time LD-query processing.

– *Location server* module which enables the calculation of position of mobile clients.

Notice that the table (Table 2) including the attributes of queries present in the server is common data accessible to all modules. In addition, a memory space called Result repository enables communication between DBMS and Result check functions.

4. Management of mobile client location-dependency

Location-dependent query processing requires take into account the current location of the client. It needs also consider proximity and/or orientation conditions. In our architecture, *Query analysis and execution parameter calculation* module accepts as input an LD-query expressed in LDQ language (see §3.2.1) and analyzes its requirements. It returns a spatial query including the client location which can be evaluated by the underlying DBMS.[3]

In this section, we focus on: (i) client localization and ii) handling proximity and orientation of objects to find (depending on client location and direction) to determine a zone for searching objects,

[3]Note that many existing DBMS are enhanced with “spatial” extensions allowing spatial queries evaluation. In this paper DBMS means spatial DBMS such as Spatial Oracle used for performance analysis (see §7).

4.1. Client localization

The location of the client at time $t0$ (i.e. the initial location) and at some time t ($t > t0$) are of interest for LD-query processing. Let *Loc*(*Q*, *x*) be a function of *Location Service* module than returns Q's client location at time x. There are different solutions to implement *Loc* function according to the targeted environment:

- *Loc*(*Q*, *t0*), the position from which the query Q was sent may be either specified directly in the MOVING PROFILE clause (thus included in the query) or provided by a localization system (such as GPS) called by the query server.
- *Loc*(*Q*, *t*): the location at a time $t(t > t0)$ may be either provided by a localization system (case 1) or estimated using client speed and direction (case 2) or using sophisticated location estimate method based on traffic conditions, weather conditions, city or region map (case 3).

$$Loc(Q,t) = \left\{ \begin{array}{lll} GPSloc(CId(Q)) & if & case1 \\ NewLoc(L*(Q), Speed(Q), Dir(Q), t*(Q), t) & if & case2 \\ AdvancedLocEstimation(\ldots) & if & case3 \end{array} \right\} \tag{1}$$

where $L*(Q)$ denotes the previous location estimate of query Q client and $t*(Q)$ the time of $L*$ calculation. *NewLoc* is a function that returns the coordinates of the new location given the previous location, client speed and direction, and moving time interval $[t*(Q), t]$. There are many techniques for location estimate function (*AdvancedLocEstimate*) design and they are out of the focus of this paper.

4.2. Object search area

As query results are location-dependent, the area (or region) enclosing the searched objects should be delimited before submitting the query to the underlying DBMS. Only objects whose location is not more than *Radius*(*Q*) far from the client location should be returned as results. The value of *Radius*(*Q*) is either specified in AT DISTANCE clause or a default value (which is a server parameter). Q's client may search objects at a given (explicit or implicit) distance from his/her:

- Initial location when the query was sent (case LocReq1[4])
  e.g. SELECT name, address
  FROM Objects WHERE AT DISTANCE (5 km)
- Future target location (case LocReq2)
  e.g. SELECT name, address
  FROM Objects WHERE MOVING PROFILE (90 km/s, EAST) AT DISTANCE (25 km)
  TARGET LOCATION REACHABLE WITHIN (150 km)
- Dynamic[5] (current) location while moving and the results are updated periodically (case LocReq3)
  e.g. SELECT name, address
  FROM Objects WHERE MOVING PROFILE (90 km/s, EAST)
  AT DISTANCE (25 km) TARGET LOCATION REACHABLE WITHIN (150 km)
  TEMPORAL CONSTRAINTS (EVERY 5 min DURING 2 h)

[4]To distinguish the location requirement types, LocReq1, LocReq2... are used.
[5]Dynamic location means the location of the client when he/she receives the results. In the sequel, "current" and "dynamic" locations are interchangeably used.

- Dynamic location while moving but the results are sent once within a fixed deadline (case LocReq4)
  e.g. SELECT name, address
  FROM Objects WHERE MOVING PROFILE (90 km/s, EAST)
  AT DISTANCE (25 km) TARGET LOCATION REACHABLE WITHIN (150 km)
  TEMPORAL CONSTRAINTS (DEADLINE 2 min)
- Dynamic location while moving toward a future target location but the results are sent once without any deadline (case LocReq5).
  e.g. SELECT name, address
  FROM Objects WHERE MOVING PROFILE (90 km/s, EAST) AT DISTANCE (15 km)

Now, let us see how to determine the search area associated with query Q, which is denoted*Area*(*Q*) according to the previously identified query types.

1. Cases LocReq1 and LocReq2: in case LocReq1 (respectively LocReq2), the search area is a circle whose center is the initial (respectively future) client location $L0$ (respectively *Lf*) and its radius is the value of *Radius*(*Q*). If an orientation (i.e. North, East. . .) is required for object selection, then a fraction of the circle is selected according to the value of *Orient*(*Q*). Formally, *Area*(*Q*) is defined as follows:

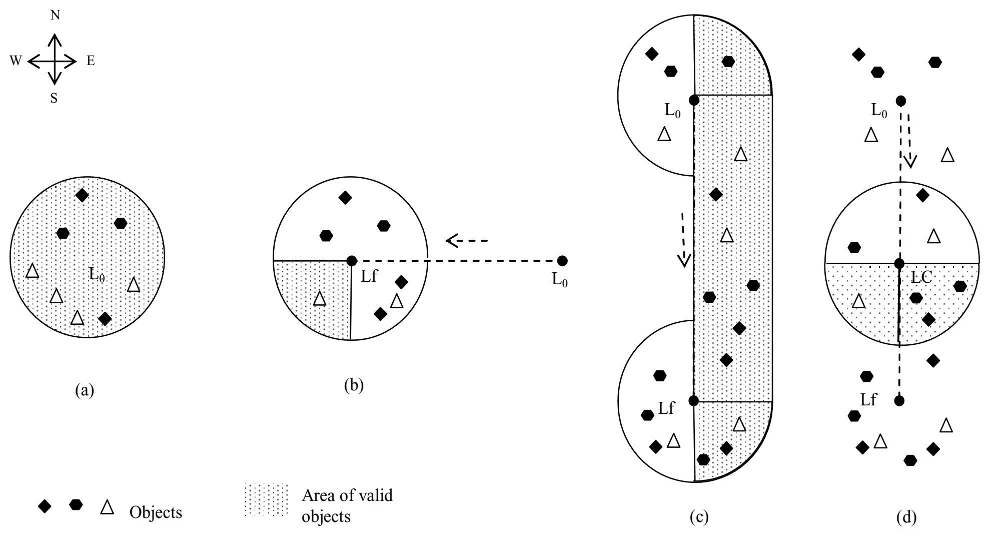
$$Area(Q) = \begin{Bmatrix} SubRegion(circle(Radius(Q), L0), Orient(Q)) & if & LocReq1 \\ SubRegion(circle(Radius(Q), Lf), Orient(Q)) & if & LocReq2 \end{Bmatrix} \tag{2}$$

   where *SubRegion*(*C*, *Orient*(*Q*)) is a function that returns a fraction (a half or a quarter) of a circle C taking into account the value specified in ORIENTATION clause. *If* no ORIENTATION clause is specified, *SubRegion*(*C*, *Orient*(*Q*)) returns C. For example if NORTH (respectively SOUTH WEST) is specified then the search region is formed by a half (respectively quarter) of circle where point ordinates are greater than or equal to the ordinate of the circle center (respectively coordinates are less than or equal to the coordinates of the circle center). Figure 3 illustrates some search area examples.
2. Case LocReq3: in this case the client requires results are periodically updated while he/she is moving from $L0$ location to *Lf* location. Our architecture includes two strategies for handling continuous queries: *Once-for-all search* and *Piecewise search*. *Once-for-all* strategy is the default strategy. The server administrator may switch continuous query management to *Piecewise* strategy either at server setup phase or dynamically using some monitoring metrics to avoid Once-for-all search drawbacks (see below). In addition, if Once-for-all strategy fails because of a tight deadline, Piecewise strategy is automatically tried to retrieve some results.

   - Once-for-all search: the search area is composed of the whole zone between $L0$ and *Lf* locations and takes into account the orientation of results (see Fig. 3c). Then the yielded area is submitted once to the DBMS Thus, *Area*(*Q*) is formally defined as:

$$Area(Q) = \begin{matrix} SubRegion(Circle(Radius(Q), L0), Orient(Q)) \cup \\ SubRegion(Plg(L0, Lf, Radius(Q)), Orient(Q)) \cup \\ SubRegion(Circle(Radius(Q), Lf), Orient(Q)) \end{matrix} \tag{3}$$

     where *Plg*(*A, B, r*) is the parallelogram formed by points: $(A_{abs}, A_{ord} - r)$, $(A_{abs}, A_{ord} + r)$, $(B_{abs}, B_{ord} - r)$, $(B_{abs}, B_{ord} + r)$. X_{abs} (respectively X_{ord}) is the abscissa (respectively ordinate) of point X. As for circle form, *SubRegion* function is applied to a parallelogram to select the area



(a) No orientation and no moving are specified,
(b) SOUTH-WEST orientation and client is moving towards WEST and future target location is *Lf*
(c) EAST orientation, client moving towards SOUTH and objects required while moving till *Lf*
(d) SOUTH orientation, client moving towards SOUTH, and objects required BEFORE entering target zone moving till *Lf*

Fig. 3. Examples of search area.

complying with the orientation specified by the client. In Fig. 3c the parallelogram is a rectangle on east side of the client.

Once-for-all strategy maybe inefficient when the period of result sending is short (thus the server will not have enough time to retrieve all the objects) or when the number of returned objects is very high (thus requiring extra-large buffer to store objects) or when the server time to process the query taking into account the whole search area from the initial to the final location is high (thus leading many other queries to miss their deadlines when they are scheduled after continuous queries with long processing time). To avoid such drawbacks, a second strategy is provided.

– Piecewise search: it consists in periodic submission of the query to the server with a limited search area taking into account the maximum distance that may be covered during a period (the period specified in EVERY clause). The obtained results are appended to those results returned by the previous executions of the continuous query. The search area is defined as follows:

$$Area(Q) = \begin{array}{l} SubRegion(Circle(Radius(Q), Lc), Orient(Q)) \cup \\ SubRegion(Plg(Lc, Lx, Radius(Q)), Orient(Q)) \cup \\ SubRegion(Circle(Radius(Q), Lx), Orient(Q)) \end{array} \quad (4)$$

where *Lc* is the current location and *Lx* is the location the client may reach during the current period:

$$Lx = LocCoord\left(Lc, \frac{Period(Q) - Cost(Q)}{Speed(Q)}, Dir(Q)\right)$$

LocCoord(*L, Dist, Dir*) is a function that returns the coordinates of location at a distance *Dist* far from location L in direction *Dir*.

The results returned by one evaluation (in case of Once-for-all strategy) or multiple evaluations (in case of Piecewise strategy) are stored in a result pool associated with query Q denoted *PoolRes*(Q).

Periodically (with a period value specified in EVERY clause), the following operations are carried out:

- *PoolRes*(*Q*) is updated by extracting and sending results relevant to the current client location and deleting those results which not spatially valid (because the objects to delete are in an area far away from the current client location).
- In case of Piecewise search, the current location is updated to yield a new search area before submitting the query to the DBMS. In other words, the search area is a sliding window.
- Submit query Q with the new search area to the DBMS. The submission of query Q is over when either the delay (specified in DURING clause) has elapsed or the distance (specified in FOR clause) is run.

3. Case LocReq4: in this case the client requires results are sent once while he/she is moving within a fixed deadline. As shown by formula (5), the search area depends on the distance to run during the fixed deadline.

$$Area(Q) = \begin{array}{l} SubRegion(Circle(Radius(Q), L0), Orient(Q)) \cup \\ SubRegion(Plg(L0, Lx, Radius(Q)), Orient(Q)) \cup \\ SubRegion(Circle(Radius(Q), Lx), Orient(Q)) \end{array} \quad (5)$$

where *Lx* is the location the client may reach before the explicit deadline expires:

$$Lx = LocCoord\left(L0, \frac{RelDeadline(Q) - Cost(Q)}{Speed(Q)}, Dir(Q)\right)$$

4. Case LocReq5: in this case, the client searches objects at a given distance around his/her current location while moving without explicit deadline or implicit deadline which could be derived from the query (e.g. "select the restaurants 500 m far from my position" or "select the gas stations 10 km far from my position, I am driving north at 120 km/h"). This type of query means that the client is expecting the results returned as soon as possible. Two strategies may be considered for handling this type of query: *best effort* strategy and *better than best effort*.

- Best effort scheduling: no deadline is assigned to the query and it is queued in a FIFO queue and evaluated only when no queries with deadlines are waiting for processing (i.e. queries with deadlines have priority over best effort queries). Such a strategy is easy to implement. However, it may result in a long waiting time before the client receives anything thus jeopardizing the result usefulness form the client point of view. When the load of the server is high, the probability of success of best effort queries (i.e. sending results within reasonable delay) becomes very low. Since our architecture aims at maximizing the success ratio of queries and providing some fairness between clients, the best effort strategy is not selected to be included in our architecture.
- Better than best effort scheduling (i.e. by assigning an artificial deadline to query): generally when a user of a computerized system looks for some information he/she does it with implicitly understanding that the response is returned within a reasonable time (i.e. the implicit system latency). Let the 'reasonable' latency[6] of the query server be *MaxServDelay*[7], which is fixed at

[6]Maximum server latency includes the maximum waiting time in queue before being processed and maximum delay for the transfer of a significant amount of results.

[7]Notice that in case of server overload or tight query requirements, the sever may be unable to find valid results for some queries whatever their type is (with or without deadline). Thus the client submits a query with the understanding that he/she will conclude that his/her query failed if he/she does not receive a response within *MaxServDelay*.

sever setup. Thus, the maximum search area is defined by the current location $L0$ and the location *Lx* that the user could reach if the results were sent taking into account the highest response time of the server.

$$Area(Q) = \begin{array}{l} SubRegion(Circle(Radius(Q), L0), Orient(Q)) \cup \\ SubRegion(Plg(L0, Lx, Radius(Q)), Orient(Q)) \cup \\ SubRegion(Circle(Radius(Q), Lx), Orient(Q)) \end{array} \tag{6}$$

$$Lx = LocCoord\left(L0, \frac{MaxServDelay - Tqt(Q)}{Speed(Q)}, Orient(Q)\right)$$

4.3. *Query rewriting and cost estimate*

Once the search area is calculated, a transformation process is achieved by Rewriting query function to yield a spatial query that the underlying DBMS is able to evaluate. For space reason we'll provide the detail of query rewriting. Let *Spatial*(*Q*) be the spatial query associated with LD-query Q.

Spatial(*Q*) is then handled by Cost estimate function, which submits the spatial query to DBMS not to process but to evaluate its cost, i.e. its estimated processing time (*Tserv*) and its amount of results [4]. From the returned DBMS cost, *Tserv*(*Q*) and *Trt*(*Q*) attributes are updated. We assume that the client profile (in particular the bit rate of his/her mobile equipment) is known to enable the estimate of query result transfer time (*Trt*).

5. Real-time constraints handling

The objective of *Timing Constraints Management* module is to maximize the percentage of queries meeting their deadlines (i.e. maximize the success ratio). It includes the following functions: calculation of deadline according to the query type, admission test, EDF scheduler, and Continuous query management. In order to maximize the success ratio, we combine a query EDF-scheduling algorithm with an admission test to early reject queries which have no chance to meet their deadlines.

5.1. *Calculation of query deadlines*

The deadline associated with a query is either explicit (i.e. determined from the *timing* parameter specified in the query) or implicit (i.e. determined from relationship between the result spatial validity and the movement of the client). To meet query deadlines, a scheduling algorithm must be used. Our architecture is based on a well-tested discipline, EDF. Recall that EDF uses absolute deadlines to give priority to queries [1]. In the sequel, the absolute deadline of query Q is denoted *AbsDeadline*(*Q*).

5.1.1. *Explicit deadline calculation*

Explicit deadlines are specified in queries that include a timing constraints requirements part. There are two cases:

1. Queries with a fixed deadline include a relative deadline (specified via DEADLINE clause). The absolute deadline must consider the absolute time of query reception (*tqr*) by the server, the relative deadline included in the query (*RelDeadline*), the estimated query transfer delay (*Tqt*), the estimated processing time of the query by the server (*Tserv*), and the time to transfer results to the client (*Trt*):

$$\begin{aligned} AbsDeadline(Q) &= tqr(Q) + RelDeadline(Q) - Cost(Q)Cost(Q) \\ &= Tqt(Q) + Tserv(Q) + Trt(Q). \end{aligned} \tag{7}$$

2. Continuous queries require the results are periodically (the period is specified via EVERY clause) updated during some time interval (either specified via DURING <time> clause or derived from the distance specified in FOR <distance> clause and client speed). As previously explained (see search area for LocReq3) our architecture includes two strategies for handling continuous queries: Once-for-all and Piecewise strategies.

   – If Once-for-all search is chosen, the query is submitted once to the DBMS and the results are buffered and then gradually transmitted to client. Consequently, there is one absolute deadline to determine:

$$AbsDeadline(Q) = tqr(Q) + Period(Q) - Tqt(Q) - Tserv(Q) - \frac{Trt(Q)}{\left\lfloor \frac{Duration(Q)}{Period(Q)} \right\rfloor} \quad (8)$$

   Tserv and *Trt* represent the estimated DBMS processing time and result transfer time for the query Q considered once. Since all the results are not sent to client in one updating period, the absolute deadline should consider the transfer time of results associated with one updating period not all the duration of the continuous query. Thus, we use the average estimated result transfer time over *Duration*(*Q*).

   – If Piecewise search is chosen, the query is periodically submitted to the DBMS with a limited search area. Thus a deadline should be associated with each periodic evaluation of query Q as shown by formula (9):

$$AbsDeadline(Q) = \begin{cases} tqr(Q) + Period(Q) - Cost(Q)\, if(FirstEvaluation) \\ AbsDeadline(Q) + Period(Q) \quad Otherwise \end{cases} \quad (9)$$

   Notice that *Tserv* and *Trt* are estimated based on a partial search area (not on the whole search area as in Once-for-all search).
   To meet the duration (specified either in DURING or FOR clause) for the evaluation of query *Q*,***AbsDeadline***(*Q*) is calculated $\lceil Duration(Q)/Period(Q) \rceil$ times.

5.1.2. Implicit deadline calculation

Here we consider queries for searching objects without explicit deadline where the client may be searching for objects around his/her:

– future target location and the results should be returned as soon as possible (case ImplictDeadl1),
– current location and the results should be returned as soon as possible (case ImplictDeadl2),
– future target location and the results should be returned before entering the zone around the target future location (case ImplictDeadl3), the clause BEFORE is used in this case,
– future target location and the results should be returned while crossing the zone around the target future location (case ImplictDeadl4), the clause CROSSING is used in this case.

The first two cases are the most used to specify queries for searching objects while moving. The other two cases are used to enable the client send very early their queries (even if they don't need immediately the results) to be early queued and then to have a very high probability of processing success (i.e. long time is left to the server to send all the valid results).

1. Cases ImplicitDeadl1 and ImplicitDeadl2: we mentioned in case LocReq5 that when no timing indication may be derived from the query, the default meaning is that the client is expecting the results within a delay not exceeding the maximum server latency (*MaxServDelay*). We also

discarded best effort scheduling of queries. Consequently, in both cases, the time between sending the query and receiving the result should not exceed *MaxServDelay*, then the deadline is as follows:

$$AbsDeadline(Q) = CurrentTime + MaxServDelay - Tqt(Q) \tag{10}$$

However, it should be noticed that the search area is not the same in both cases: search area is centered around the future target (respectively current) location in case ImplicitDeadl1 (respectively ImplicitDeadl2).

2. Cases ImplicitDeadl3 and ImplicitDeadl4: in both cases more time is given to the server to return the results so larger[8] deadlines are yielded. The deadline consider the time to reach the future target location and if the results should be received by the client before entering or while crossing the future target area. Thus the deadline is defined as follows:

$$\begin{aligned} &AbsDeadline(Q) = tqr(Q) + MovingTime - Cost(Q) + Margin \\ &Margin = \begin{cases} -Radius(Q)/Speed(Q)\; if\; LoZ(Q) = BEFORE \\ Radius(Q)/Speed(Q) \quad if\; LoZ(Q) = CROSSING \end{cases} \end{aligned} \tag{11}$$

where *MovingTime* = *TimeToGo*(*Q*) if REACHABLE WITHIN <time> clause is specified,
MovingTime = *DistToGo*(*Q*)/*Speed*(*Q*) if REACHABLE WITHIN <distance> clause is specified,
MovingTime = *Length*(*L0,FixLoc*)/*Speed*(*Q*) if AT <fixed_coordinates> clause is specified
and *Length*(*L1, L2*) is a function that returns the distance between two locations$L1$and$L2$.

5.2. Admission test and scheduling

Our architecture is based on EDF, a well-known and well-tested algorithm and commonly used in real-time systems. Queries are assigned absolute deadlines and inserted (if they pass admission test) in an ordered Read-query queue according to their deadlines: the query with the lowest deadline is at queue head (i.e. the first position) and the query with the highest deadline at queue tail (i.e. the last position). Notice that continuous query handling requires that a deadline is assigned to each instance (or activation) of each continuous query. A function called Continuous query manager included in *Timing constraint management* module evaluates periodically deadlines and removes continuous queries when their evaluation time interval (specified in EVERY or FOR clause) is exceeded.

5.2.1. Admission test

The absolute deadline *AbsDeadline*(*Q*) and the estimated DBMS processing time *Tserv*(*Q*) determined by Deadline calculation and Cost estimate functions are used to test if query Q may be processed by DBMS on time. Admission test considers the queries already admitted and still in the ready query queue.

1. Non-continuous query admission test: Query Q is admitted for evaluation by DBMS if 1) its deadline will be guaranteed[9] and 2) the guarantee of deadlines of queries already accepted will not be affected. In other words, query Q is admitted if it may be inserted in Ready-query queue at some position such that Q's deadline will be meet and the deadlines of already accepted queries

[8]Notice that since an EDF-based query scheduled is used in our architecture, when there no queued queries with tight deadlines, the server may evaluate and send the results earlier even if the absolute deadline derived (for queries using BEFORE and CROSSING clauses) is high.

[9]Notice that the guarantee of query deadline is not absolute since it depends on the accuracy of cost estimate (DBMS processing time and result transfer time estimate) used to calculate the deadline.

with deadlines higher or equal to Q's deadline will be meet. To accept query Q, the work load (i.e. the sum of query DBMS-processing times) of ready queries with lower deadlines is considered. Formally, the admission test is as follows:

$$AdmissionTest(Q) = \begin{cases} True \quad if \quad \forall j\ k \leqslant j \leqslant Nq\ AbsDeadline(IdQ(j)) \geqslant \begin{pmatrix} tc + Tserv(Q) + \sum_{i=1}^{j} Tserv(IdQ(i)) \\ + \sum_{i \in UPQ} Tserv(IdQ(i)) \end{pmatrix} \\ False\ otherwise \end{cases} \tag{12}$$

where: k is the index -in Ready-query queue- of the first query of which the deadline is greater than *AbsDeadline*(*Q*). *Nq* is the number of queries in Ready-query queue. *IdQ*(*j*) is a function that returns the identifier of query at position j in Ready-query queue. *tc* is the current time. *UPQ* is the identifier set of queries being processed by DBMS (this set is updated by query scheduler whenever a query is submitted to DBMS and when its processing is finished). $\sum_{i \in UPQ} Tserv(IdQ(i))$ is used in order to consider the worst case where several queries have been submitted to be processed in parallel and they still under processing at DBMS level.
If *AdmissionTest*(*Q*) if true, query Q is inserted in Ready-query queue at position k otherwise it is rejected.

2. Continuous query admission test: the previous scheme is adequate for queries that are evaluated once. Admission test of continuous queries is a bit complex: should the admission test be carried out once for the continuous query as a whole or for each query instance (i.e. for each periodic execution of a continuous query)? There are pros and cons for each solution.

   – One test: it avoids periodic tests and provides the same (constant) level of guarantees for the client (i.e. either the query is accepted for all time interval specified in DURING or FOR clause or it is rejected). However, this scheme has two drawbacks: difficulty of implementation (i.e. all the instances of the query should be included in Ready-query queue) and unfairness (i.e. some continuous queries may monopolize server capacity).
   – Periodic test: this scheme has the inverse properties the previous one: more fairness, but no guarantee that the client will periodically receive updated results.

Our architecture includes both solutions but one of them must be selected at server setup.[10]

5.2.2. Query scheduling

EDF query scheduler function is in charge of submitting queries to the underlying DBMS. There two strategies for submitting queries:

– When DBMS becomes idle, the query at the head of Ready-query queue is extracted and submitted to DBMS. Query scheduler waits till DBMS becomes idle again to submit the next query.
– Multiple queries (taking into account their deadlines) are submitted in parallel to DBMS. Query scheduler does wait till the DBMS to submit a new query.

The first strategy is easy to implement. However, it suffers low performance. It is well known that DBMS response time includes CPU time and disk access time. If DBMS has only one query to process

[10]One may think of a more powerful mechanism that dynamically switches between both solutions according the server load or other metrics.

it remains blocked during the I/O operations. To achieve better performance, we propose to use the second solution, i.e. submit several[11] queries which will be processed in parallel at DBMS level. The second solution achieves good performance particularly when the average number of waiting queries is high. However, it should be noticed that when a set of queries are submitted to DBMS to process in parallel, the DBMS-processing order (scheduling) of this set queries is not visible to the query submitter. Consequently, the selection of queries to submit in parallel to DBMS considers the deadlines of selected queries to avoid priority inversion leading to deadline missing. For example let us consider that at time $t = 100$, three queries are ready $Q1$, $Q2$ and $Q3$. $Q1$ (respectively $Q2$ and $Q3$) has a deadline equal to 110 (respectively 120 and 125) and an estimated processing time equal to 5 (respectively 4 and 30). $Q1$ and $Q2$ may be submitted together since their deadlines are met whatever the order they are processed by DBMS. However, $Q3$ can not be submitted in parallel neither with $Q1$ nor with $Q2$.

Another issue dealt with by the query scheduler is the monitoring of ready-query queue. Query deadlines assigned based on cost estimate of queries. Unfortunately the accuracy of estimate depends on the underlying DBMS and the types of SQL-like queries yielded from LD-queries. In case of under-estimate of DBMS processing time for some queries, DBMS finishes processing some queries later than expected (by estimate). Consequently, some other queries in ready queue either have their deadlines missed while they still in the queue or will not have sufficient time to be processed on time once when submitted to DBMS. To avoid the execution late queries, query scheduler checks the deadline of the query at ready queue head before submitting it to DBMS (i.e. if the deadline is already missed or if the remaining time till query deadline is less than the estimated DBMS processing time, the query is rejected).

6. Validity check and management results

Once the DBMS finishes processing a query Q the yielded results are appended to result repository. Then according to type of query the results are checked are treated as follows:

– Case of non-continuous queries: the spatial and temporal validity of results is checked. Only spatially and temporally valid objects are immediately sent to the client. At the end of query Q execution all results relating to query Q are removed from Result repository.
– Case of continuous queries: the spatial validity of the new results is checked to be appended or not to results pool of query. Notice that the results of continuous queries are sent periodically not necessarily just after their production. Continuous query result management function is in charge of the periodic transmission of results to clients. Given the period of query Q and its result pool, denoted *ResPool*(*Q*), this function extracts the temporal and spatial valid results and transmits them to client. In addition, each result which is obsolete is removed from *PoolRes*(*Q*) and deleted.

Whatever the type of query is, if partial results are searched (i.e. when PARTIAL RESULTS clause is specified), only an amount of results that fulfills[12] AT LEAST or AT MOST requirements are considered for transmission to the client. In the architecture presented in this paper, partial result requirements are used only to speed the transmission of results to client in case of AT LEAST requirement. In case there

[11]The maximum number of queries being processed by DBMS is a server parameter fixed at setup.
[12]When the number of valid results returned by DBMS is higher than the one expressed in AT LEAST or AT MOST clause, the objects transmitted to the client may be selected either randomly among valid objects or according to some criteria (e.g. distance). In our architecture, the closest objects with regard to the current client location are considered as a priority.

is not enough time to transmit all the results returned by the DBMS, then only a subset of objects are sent. One may think of more adaptive query management techniques to adjust the relative deadline and search area according to the (maximum or minimum) number of objects required by client.

The inaccuracy of estimating deadlines and client location may result in returned objects that are not valid either because the estimate location of the client is not (approximately) the same than the real location when the client will receive the results or because the actual delay to return the results is higher than the estimated delay. To consider estimate inaccuracy, *Result check and management* module includes functions to check spatial and temporal validity of results.

6.1. Spatial validity check

Each object produced by DBMS is accompanied by its location (coordinates). The spatial validity check consists in retrieving the current location of the client and comparing such a location to the locations of each returned object. Only objects of which the distance far from current client location is less than or equal to *Radius*(*Q*) are selected. We assume that client location change during result transfer is negligible[13] (i.e. the distance run by the mobile client between the time the spatial validity check is achieved and the time the results are received by the client is negligible).

Let *tc* be the current time (time just before sending results to client) and *Results*(*Q*) the results associated with query Q. Spatial validity check of results means the extraction of a subset of results *SpaceValidRes*(*Q*) such that:

$$SpaceValidRes(Q) = \{\forall O_i | O_i \in Results(Q), Dist(LocObj(O_i), Loc(Q,tc)) \leqslant Radius(Q)\} \quad (13)$$

where $LocObj(O_i)$ is a function that returns the location of object O_i. This function relies on the object coordinates included by DBMS in the results. $Dist(L1, L2)$ returns the distance between two locations $L1$ and $L2$ of which coordinates are known. Notice that we consider only direct distance, i.e. the segment length between two points. If we consider roads crossing mountains, the direct distance is not relevant. Function *Dist*() should be designed accordingly (i.e. taking into account the road map).

6.2. Temporal validity check

After the result spatial validity check, a second check is carried out in order to reject produced results if they missed the deadline estimated by Deadline calculation function.

Once the results are produced their size (in bytes) is known. Thus, the result transfer time should be calculated with more accuracy than as done by Cost estimate function run before query execution. Let *Trt'*(*Q*) denote transfer time calculated by Temporal validity check function. If partial results are required, then *Trt'*(*Q*) is calculated accordingly (i.e. *Trt'*(*Q*) is the time associated with the transfer of the whole, the minimum or maximum results). Then we can formulate the temporal validity check as follows:

$$TemporalValidRes(Q) = \left\{\begin{array}{lll} True\ if & tc + Trt'(Q) \leqslant AbsDeadline(Q) & \wedge TrtQ) > Trt'(Q) \\ True\ if & tc + Trt'(Q) \leqslant AbsDeadline(Q) - (Trt'(Q) - Trt(Q)) & \wedge TrtQ) \leqslant Trt'(Q) \\ False\ otherwise & & \end{array}\right\} \quad (14)$$

[13] It is a realistic assumption if one takes into account the speed (in km/s) of mobile clients and the speed (in bits/s) of modern networks.

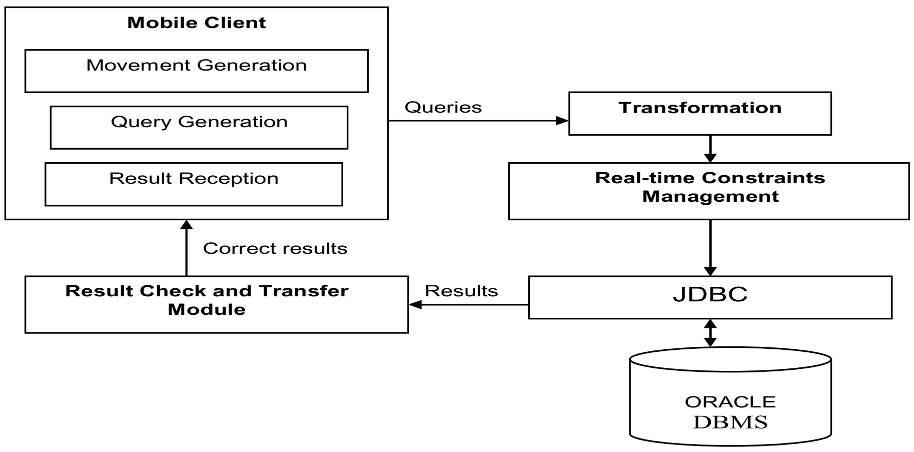


Fig. 4. Prototype architecture.

where *tc* is the current time.

It should be noticed that temporal validity check is not applied to continuous queries since the produced results are stored in result pools and are sent periodically to mobile clients and obsolete objects (i.e. those objects too far from the last known client position) are deleted by Spatial validity check function.

7. Performance evaluation

We implemented a prototype to validate the feasibility of our proposals and to evaluate the performances of a subset of proposed methods for LD-query processing. We essentially investigated the benefits of checking spatial validity before data transfer and considering real-time constraints while processing LD-queries.

7.1. Experimentation prototype environment

We implemented different modules of our architecture on top of Oracle (Release 10.1.0.2 standard edition). An extension called Oracle Locator is provided in order to allow the use of geometry types and spatial operators [13]. *Java* libraries are also provided by Oracle in order to handle spatial objects. Thus, our prototype is developed in *Java* language (*Java Development Kit JDK 1.4.2*). The experiments were performed on a *Sun Blade 150* workstation (550 MHz processor speed, 300 MB memory size) communicating with a *Sun Fire V240* workstation (1.5 GHz processor speed, 8 GB memory size) on which Oracle DBMS is setup. Both workstations are connected to 100 Mb/s Ethernet network.

The architecture of the prototype (Fig. 4) follows the same principle as the generic architecture (Fig. 2). Indeed, modules are implemented on top of Oracle DBMS in order to take into account mobile client location-dependency of the query results and real-time constraints. The JDBC Application Programming Interface is used to connect the developed java programs to Oracle database. In our tests, the Oracle cache is active. However, it has no contribution because all submitted queries are independent in order to test all LD-query types. Hence, no result is cached and all submitted queries are completely processed by Oracle. Consequently, the processing time of LD-query is increased and the respect of deadline is more complex. This configuration enables evaluate our prototype in the worst case.

Table 3
Relation cardinality and object density

Relation	Number of tuples	Object density (Obj/km^2)
R_1	5000	5
R_2	5000	50
R_3	20000	5
R_4	30000	5
R_5	10000	5
R_6	10000	50

We also developed a module to simulate client mobility. We consider mobile clients as moving points having initial locations uniformly distributed in a square area (denoted as *Work_Space*). Locations of moving objects and clients are represented in the WGS 84 reference system (World Geodetic System). This reference system is used by GPS. The sub-module *Movement Generation* randomly generates the initial position (longitude, latitude), the speed and directions of each client. We assume that the speed and direction are constant between the moments of query sending and result receiving. The *Result Reception* sub-module allows the receipt and display of results produced after query processing. To compute the result transfer time, we assume that the wireless network bandwidth is 171.2 kbps (like the maximum GPRS bandwidth) [17].

We tested this prototype with several LD-query examples querying data related to 400 French hospitals and we checked on a map the query results. In the following subsection we present the queries and data used to conduct performance evaluation using the implemented prototype.

7.2. Data and query models

We assume that the LD-queries are composed by proximity operator and one, two, three or four join operations. Here, we use the proximity operator[14] SDO_WITHIN_DISTANCE (*ObjLocAttribute, query_dist*). The value of *query_dist* is randomly generated between 0.2 km and 10 km. The number of tuples of the operand relations (denoted $||R||$) varies randomly between 5000 and 30000. The join selectivity factor (R_i, R_j) is $JSF = 1.5/Max(||R_i||, ||R_j||)$ [22]. The queried data is randomly generated. This data describes different sets of fixed objects (e.g. name, address, longitude and latitude of hospitals or pharmacies). To compute the distance between two points in the WGS 84 reference system (e.g. for result spatial validity check), we implemented an algorithm based on the *Vincinty's* formula [26]. The same formula is used by Oracle Locator. Certain relations $Robj_i$ represent a set of target objects Obj_i located inside an area called work space area (denoted $Work\_Space_i$). The locations of the target objects are uniformly distributed in the area $Work\_Space_i$. The number of target objects of each relation and the size of the $Work\_Space_i$ are chosen in such a way to have different target object densities ($DObj_i$) (cf. Table 3). For example, rare objects like hospitals have low density (e.g. 5 objects/km^2) and frequent objects like fire hydrant have high density (e.g. 50 objects/km^2).

As seen on Table 4, depending on number of join operations and target object densities we test four query forms.

The query arrivals (denoted as A) are generated according to a Poisson process with mean arrival rate λ (queries/second) varying from 0.1 to 1 query by second. The deadline associated with query (denoted as D) is set using the following formula [30]: $D = A + (1 + Slack)*E$, where E is an estimate of

[14]SDO_WITHIN_DISTANCE is an operator of Spatial Oracle, which is used to select objects at a given distance. This operator is equivalent to AT DISTANCE clause used in LD-query specification language (§3.2.1).

Table 4
Experimented queries

Req 1:	Select R1.b, R3.b
	FROM R1, R3
	WHERE SDO_WITHIN_DISTANCE (R1.LocObj$_1$, 1.5 km)
	And R1.a = R3.a
		Req 2:	Select R2.b, R3.b
			FROM R2, R3
			WHERE SDO_WITHIN_DISTANCE (R2.LocObj$_2$, 1.5 km)
			And R2.b = R3.b

Req 3:	Select R5.b, R3.b FROM R5, R4, R3, R2, R1 WHERE SDO_WITHIN_DISTANCE (R5.LocObj$_5$, 1.5 km) And R5.a = R4.a And R4.b = R3.b And R3.c = R2.c And R2.d = R1.db	Req 4:	Select R6.b, R3.b FROM R6, R3, R2, R1 WHERE SDO_WITHIN_DISTANCE (R6.LocObj$_6$, 1.5 km) And R6.a = R4.a And R4.b = R3.b And R3.c = R2.c And R2.d = R1.d

response time plus transfer time of a query Q. The slack factor *Slack*, is a uniformly distributed variable in $[-1,10]$. Notice that slack factor can be negative. In this case, the query has no chance to meet its deadline. This may happen when the user specifies very short deadlines.

7.3. Result analysis

First, we discuss the benefits of checking the spatial validity of results. Secondly we evaluate the impact of considering real-time constraints on the percentage of queries which miss their deadlines and on incorrect result percentage.

7.3.1. Benefits of spatial validity check

We conducted experiments to identify the parameter impacting the percentage of incorrect results and to evaluate the overhead and the communication cost optimization provided by the spatial validity check procedure.

1. Parameters influencing query result spatial validity
   For each query (cf. Table 4), we varied the query client movement speed and we measured the percentage of incorrect target objects (with regard to client position when results are received). We obtained the results showed on Fig. 5.
   As observed from Fig. 5, the movement speed, the number of join operations and the target object density have an impact on the incorrect result percentage. This percentage remains relatively low when the query includes only one join and the target object density is low (*Req 1*). This percentage is 0.28% for a client speed less than 50 km/h and remains lower than 2.5% even for high client speed. Indeed, when the query is basic (e.g. including only one join like *Req 1*) the response time is low. So a small distance is run before receiving query results. Since there are a few objects, a few results would be incorrect. This percentage becomes higher when target object density is 50 objects/km^2 and the number of joins is 4 (*Req 4*) (11.7% when client speed is higher than 110 km/h). It should also be noticed that the percentage of incorrect results increases slowly when client speed does not exceed 110 km/h and when the target object density is low (*Req 1* and *Req 4*). We also observe that the gap (in terms of incorrect result percentage) is higher when the target object density is varied (e.g. *Req 1* versus *Req 2*) in comparison to when the join number is varied (e.g. *Req 1* versus *Req 3*). Although the number of joins is higher for query *Req 3*, the percentage of incorrect results is lower than *Req 2* (including only one join but with higher object density). Hence we can state that the target object density has more impact on incorrect result percentage than the number of joins.

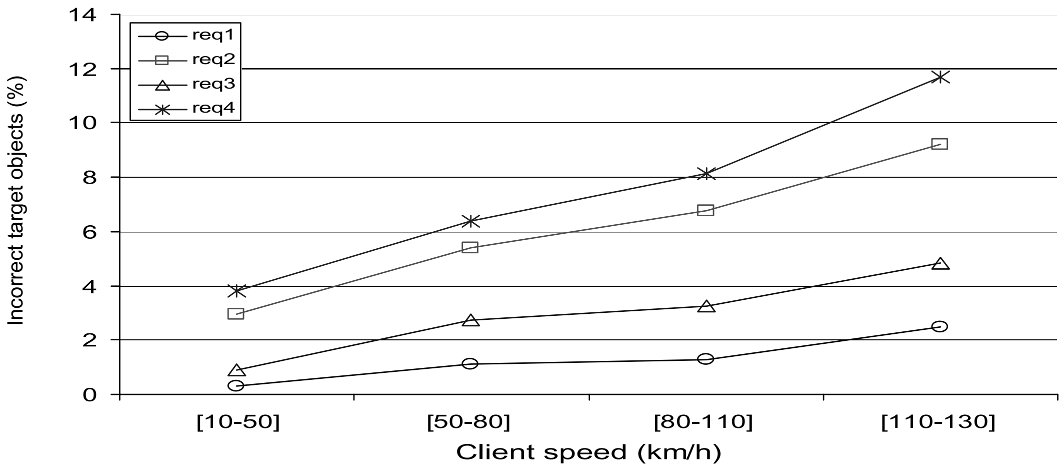


Fig. 5. Percentage of incorrect results.

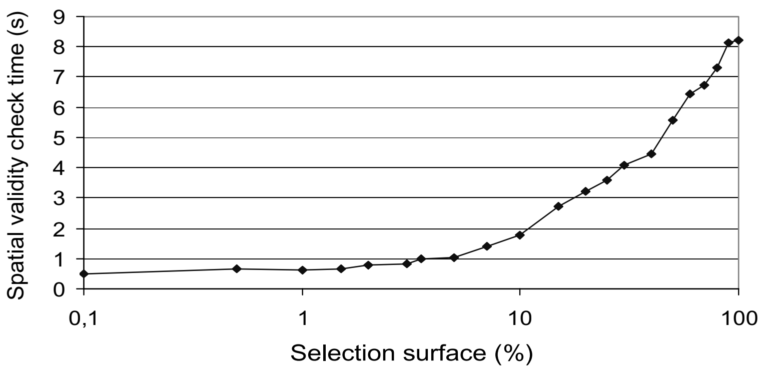


Fig. 6. Spatial validity check time.

2. Spatial validity check overhead and communication cost optimization
   We varied the selection surface from 0.1% to 100% of the work space area and we measured the spatial validity check time *TSpCheck*. We conducted these experiments with query *Req 4*. We also assume that the client movement speed is greater than 110 km/h.
   Since the spatial validity check method computes the distance between the current client location and each target objet returned by the underlying DBMS, *TSpCheck* depends on the number of returned target objects. This number is higher when the selection surface is larger (the target objects are assumed to be uniformly distributed on the work space area). This is why -as seen from Fig. 6 the spatial validity check time increases as the selection surface increases. For example, *TSpCheck* is more than 8 seconds when the selection surface is equal to all the work space area (all target objects are returned by the DBMS) and it is lower than 650 milliseconds for a selection surface less than 1% of the work space area. We also measured the query response time without spatial validity check which we compared with query response time with spatial validity check. The results showed that the spatial validity check introduces low overhead (always less than 3% whatever the selection surface is). In fact, although the spatial validity check takes time, it reduces transfer time by avoiding the transfer of incorrect results (Fig. 7). Therefore the response time

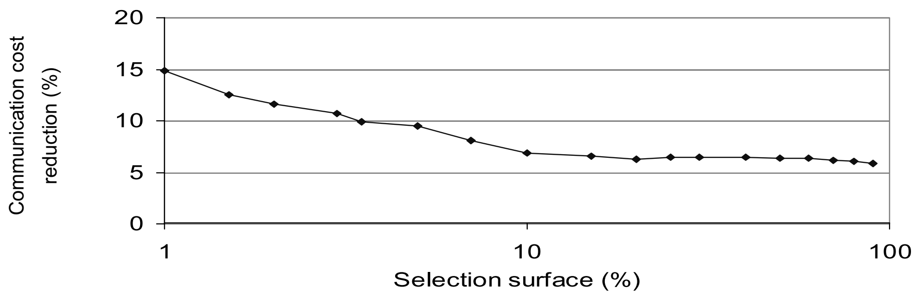


Fig. 7. Communication cost reduction.

(including transfer time) is reduced when the spatial validity check is performed. We measured the communication cost reduction, CCR, using the following formula: $CCR = (1 - \frac{DSpCheck}{DNSpCeck}) \times 100$

Figure 7 shows that the communication cost reduction decreases slowly from 15% to 5.5% as the selection surface increases if we do not take into account the limit cases (i.e. very small selection surface equal to 0.1% of work space area or selection equal to all the work space area). This is due to the fact that when the selection surface is small, even a short movement of the client, between the moments of sending the query and receiving the results, may have a significant impact on the incorrect result percentage. This is not the case when the selection surface is very large.

To sum up, we observed that the percentage of incorrect results is affected by several parameters: client speed, join number, selection surface and the target object density. Thus, checking the result spatial validity is not efficient in all cases. Indeed, in certain case this method introduces an overhead without significantly reducing the communication cost. For example, when all the clients are in the city centre (low speed), the target objects are rare (low density) and the selection surface is large, the percentage of incorrect results would be very low. In this case, the communication cost reduction would also be very low. In other cases (e.g. high target object density and small selection surface), this method can reduce the communication cost by 15%. Notice that certain parameters can be known before designing location-based applications (e.g. target object density). So the application designer can decide to implement (or not) result spatial validity check method depending on the accepted incorrect result percentage and on the identified parameter values (e.g. target object density, expected mobile client behaviors).

7.3.2. Benefits of considering real-time constraints module

To evaluate the benefit of using the *Timing Constraints Management* module we compared the missed deadline percentage (denoted as MDP) obtained using a version of the prototype implementing this module (denoted as RCM: real-time constraint management) with a version of the prototype not implementing this module (denoted as NRCM).

1. Percentage of queries missing their deadlines
   In this experiment we varied the mean query arrival rate and we measured the MDP for both versions (RCM and NRCM). Notice that for the NRCM prototype version, the FIFO scheduling is used to serve arriving queries. The results are depicted on Fig. 8.
   We observe that the *Real-time Constraints Management* module reduces the percentage of queries which miss their deadlines, especially when the mean query arrival rate exceeds 0.4. Indeed, at this rate, the percentage of missed query deadlines is less than 30% when the *Real-time Constraints Management* module is used, while this percentage is more than 85% in the other cases. This is due to the fact that we implemented the real-time scheduling algorithm EDF which gives higher

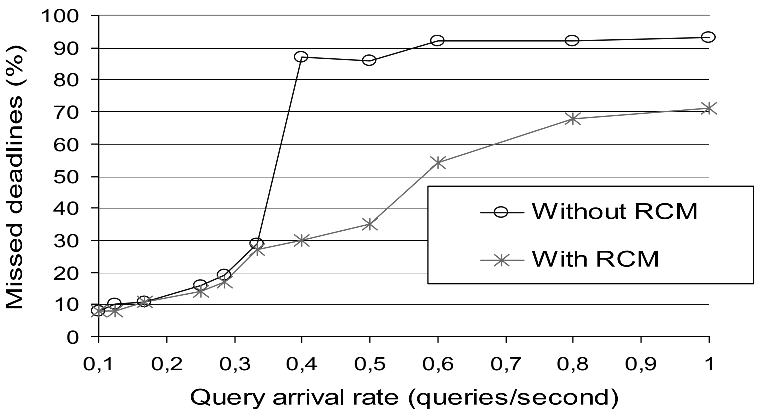


Fig. 8. Percentage of queries missing their deadlines with and without using RCM module.

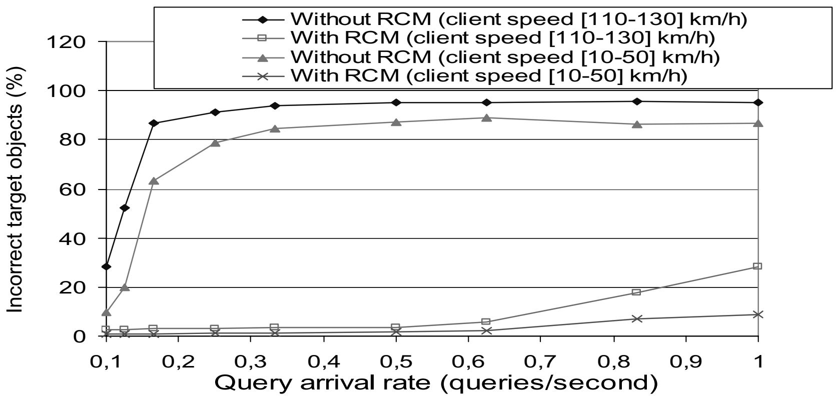


Fig. 9. Percentage of incorrect results with and without using RCM module.

priority to queries with the closest deadlines. This algorithm is efficient when the mean arrival time is not very high. It should be noticed that, although the MDP remains low using RCM, the MDP is efficient in both cases when mean arrival time is high (71% of queries miss their deadline using RCM and 94% with NRCM when one query per second is received by the server).

2. Percentage of incorrect query results
   In this experiment, we varied the query arrival rate and we measured the percentage of incorrect query results. First, we assumed that the mobile clients have a low speed (less than 50 km/h) and we compared the performances obtained with and without using the *Real-time Constraints Management* module. Then we conducted the same experiment and we assumed that the mobile clients move at high speed (110 km/h). In this experiment we assumed that all query have implicit deadlines.

Figure 9 shows that the incorrect result percentage increases rapidly and it quickly becomes very high in the absence of *Real-time Constraints Management* module. In fact, waiting time in the queue increases when the arrival rate is high. Hence, the query response time may be high. So, the user may go through an increased distance before receiving the query results. Thus, several clients may get the results after being very far from their location when their queries are sent. When the *Real-time Constraints Management* module is used, a deadline is derived depending on client movement and specified distance (radius of selection surface). The EDF algorithm gives higher priority to queries with closest deadlines. Consequently, clients moving at high speed searching target objects within a short distance are favored. In addition, if during waiting time in ready-query queue the query deadline expires, the query is rejected. So the DBMS does not process queries the results of which are definitely incorrect. These reasons justify an incorrect result percentage which is considerably low when the *Real-time Constraints Management* module is used (RCM). This is true even when the mobile client moves at a low speed.

According to these results, it may be observed that implementing the *Real-time Constraints Management* module reduces not only the missed deadline percentage, but also the percentage of incorrect LD-query results. In fact, determining an implicit deadline depending on mobile client speed and surface selection radius, and considering these deadlines while scheduling queries significantly reduces this percentage. So, it is important to take into account real-time constraints even when user does not explicitly specify a deadline.

8. Conclusion

In this paper we propose a solution for considering timing constraints while processing location-dependent queries. We propose methods to take into account client location, proximity and orientation conditions. To maximize the percentage of location-dependent queries meeting their deadlines, we use EDF scheduling algorithm jointly with an admission test. In order to avoid transmitting incorrect results (due to client location change after sending the query) and obsolete results (because of deadline expiration) we propose to check result spatial validity and result temporal validity just before data transfer. Validity check also reduces the communication cost.

We implement the modules of our architecture on top of Oracle DBMS to validate our proposal. The developed prototype is used to evaluate the performances of the proposed methods, in particular methods for spatial validity check and timing constraint handling. The results of our experiments are encouraging and show that the proposed methods reduce communication cost, increase the percentage of queries meeting their deadlines and also the percentage of correct results transmitted to users.

Our future work will mainly focus on:

1. Extending performance evaluation to other modules of the architecture and considering other types of queries (e.g. continuous location-dependent queries, partial result location-dependent queries).
2. Improving LD-query processing methods using an applicative cache in order to avoid repetitive processing of the similar sub-queries. The goal is not to cache the result of a LD-query -because the location differs for each query- but to cache repetitive sub-part of queries.
3. Considering more realistic client moving profiles where speed and direction change dynamically or where user is moving in a city. Notice that realistic profiles make location prediction more complex.
4. Using more complex data model for representing moving objects and accordingly adapt our architecture. Our objective is to anticipate the movement of the mobile objects to be able to predict their future location.
5. Investigating the use of other scheduling policies by taking into account, for example, query importance. Indeed, in many scenarios, the mobile queries may have different importance levels. For example, for medical emergency team, some situations are not critical for the patients while others can be critical.
6. Handling network characteristics and changes. One objective is to efficiently estimate data transfer and latency time and to adapt accordingly query processing.

7. In the architecture presented in this paper, partial result requirements are used only to speed the transmission of results to client. In case there is not enough time to transmit all the results returned by the DBMS, then a subset of objects are sent. One may think of more adaptive query management techniques to adjust the relative deadline and search area according to the (maximum or minimum) number of objects required by client.

References

[1] R.K. Abbott and H.G. Molina, Scheduling Real-time Transactions: a Performance Evaluation, *14th International Conference on Very Large Data Bases* (1988), 1–12.

[2] S. Bouzefrane, B. Sadeg and L. Amanton, Soft Real-Time Transactions Scheduling in a Wireless Environment, *4th International Symposium on Object-Oriented Real-Time Distributed Computing* (2001), 327–334.

[3] J. Chen, D.J. DeWitt, F. Tian and Y. Wang, NiagaraCQ: A Scalable Continuous Query System for Internet Databases, *ACM International Conference on Management of Data* (2000), 379–390.

[4] W. Du et al., Query optimization in heterogeneous DBMS, *18th Very Large Data Bases Conference* (1992), 277–291.

[5] R.H. Güting et al., A Foundation for Representing and Querying Moving Objects, *ACM Transactions on Database Systems* **1** (2000), 1–42.

[6] H. Hu, M.i Zhu and D.L. Lee, Towards Real-time Parallel Processing of Spatial Queries, *International Conference on Parallel Processing* (2003), 565–572.

[7] T. Imielinski and B.R. Badrinath, Querying in Highly Mobile Distributed Environments, *18th International Conference on Very Large Data Bases* (1992), 41–52.

[8] S. Ilarri, E. Mena and A. Illarramendi, Location-Dependent Queries in Mobile Contexts: Distributed Processing Using Mobile Agents, *IEEE Transactions on Mobile Computing* **8** (2006), 1029–1043.

[9] J. Jayaputera and D. Tanian, Defining Scope of Query for Location-Dependent Information Services, *International Conference on Embedded and Ubiquitous Computing* (2004), 366–376.

[10] E. Kayan and Ö. Ulusoy, Real-Time Transaction Management in Mobile Computing Systems, *6th International Conference on Database Systems for Advanced Applications* (1999), 127–134.

[11] J. Lindstrom, Performance of Distributed Optimistic Concurrency Control in Real-Time databases, *7th International Conference on Information Technology* (2004), 243–252.

[12] N. Marsit, A. Hameurlain, Z. Mammeri and F. Morvan, Query Processing in Mobile Environments: A Survey and Open Problems, *1st Int. Conference on Distributed Frameworks for Multimedia Applications* (2005), 150–157.

[13] Oracle Corporation, Oracle Spatial User's Guide and Reference, 10g Release 1 (10.1), Oracle documentation, 2003.

[14] K. Ramamritham, Real-Time Databases, *Distributed and Parallel Databases* **2** (1993), 199–226.

[15] K. Ramamritham, S.H. Son and L.C. Dipippo, Real-Time Databases and Data Services, *Real-Time Systems* **2–3** (2004), 179–215.

[16] P. Rigaux, M. Scholl, L. Segoufin and S. Grumbach, Building a Constraint-Based Spatial Database System: Model, Languages, and Implementation, *Information Systems* **6** (2003), 563–595.

[17] S. Sharma, I. Baek and T. Chiueh, OmniCon: a Mobile IP-based Vertical Handoff System for Wireless LAN and GPRS Links, *Software Practical Expert* **7** (2007), 779–798.

[18] A.Y. Seydim and M.H. Dunham, A Location Dependent Benchmark with Mobility Behavior, *International Database Engineering and Applications Symposium* (2002), 74–85.

[19] A.Y. Seydim, M.H. Dunham and V. Kumar, An Architecture for Location Dependent Query Processing, *12th International Workshop on Database and Expert Systems Applications* (2001), 549–555.

[20] G. Sivaradje, R. Nakkeeran and P. Dananjayan, A Prediction Based Flexible Channel Assignment in Wireless Networks using Road Topology Information, *3rd International Conference on Advances in Mobile Multimedia* (2005), 107–118.

[21] A.P. Sistla, O. Wolfson, S. Chamberlain and S. Dao, Modeling and Querying Moving Objects, *IEEE International Conference on Data Engineering* (1997), 422–432.

[22] E.J. Shekita, H.C. Young and K.L. Tan, Multi-Join Optimization for Symmetric Multiprocessors, *Very Large Data Base Conference* (1993), 479–492.

[23] M. Thilliez and T. Delot, Evaluating Location Dependent Queries Using ISLANDS, *3rd International School and Symposium on Advanced Distributed Systems* (2004), 125–136.

[24] G. Trajcevski, H. Ding and P. Scheuermann, Context-aware Optimization of Continuous Range Queries Maintenance for Trajectories, *ACM Workshop on Data Engineering for Wireless and Mobile Access* (2005), 1–8.

[25] G. Trajcevski, O. Wolfson, K. Hinrichs and S. Chamberlain, Managing Uncertainty in Moving Objects Databases, *ACM Transactions on Database Systems* (3) (2004), 463–507.

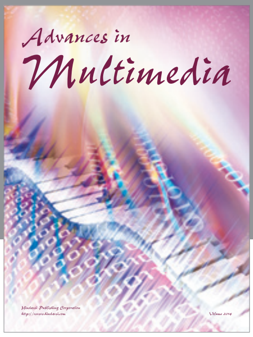
[26] T. Vincinty, Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations, *Survey Review XXII* **176** (1975), 88–93.
[27] O. Wolfson, S. Chamberlain, K. Kalpakis and Y. Yesha, Modeling Moving Objects for Location Based Services, *Developing an Infrastructure for Mobile and Wireless Systems NSF Symposium* (2001), 46–58.
[28] O. Wolfson and H. Yin, Accuracy and Resource Consumption in Tracking and Location Prediction, *Symposium on Spatial and Temporal Databases* (2003), 325-343.
[29] M. Xiong, S. Han and K.Y. Lam, A Deferrable Scheduling Algorithm for Real-Time Transactions Maintaining Data Freshness, *26th IEEE International Real-Time Systems Symposium* (2005), 27–37.
[30] M. Xiong et al., Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics, *IEEE Transactions on Knowledge Data Engineering* **5** (2002), 1155–1166.
[31] B. Zheng, W.C. Lee and D.L. Lee, On Semantic Caching and Query Scheduling for Mobile Nearest-Neighbor Search, *Wireless Networks* **6** (2004), 653–664.
[32] B. Zheng, J. Xu and D.L. Lee, Cache Invalidation and Replacement Strategies for Location-Dependent Data in Mobile Environments, *IEEE Transaction on Computers* **10** (2002), 1141–1153.
[33] J. Zhang et al., Location-based Spatial Queries, *ACM SIGMOD International Conference on Management of Data* (2003), 443–454.

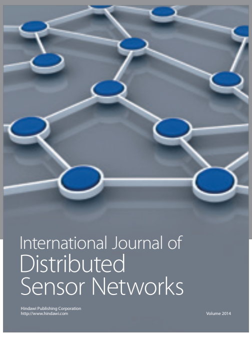
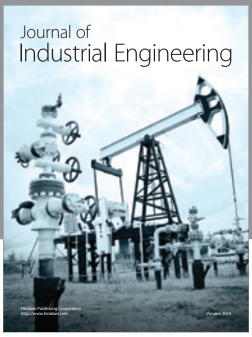
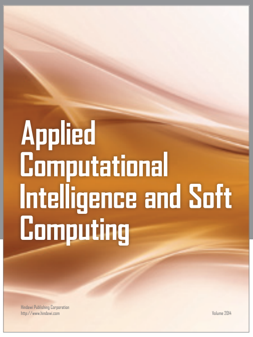
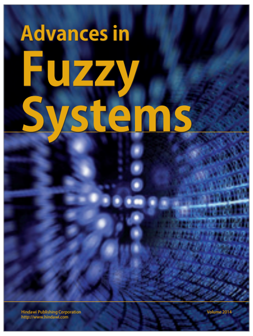
Zoubir Mammeri received his M.S, Ph.D and HDR in computer science from National Polytechnic Institute of Lorraine (France). Since 1998, he is a full professor at Paul Sabatier University of Toulouse. His research interests include: quality of service, QoS-based routing, packet scheduling, mobile ad hoc networks, sensor networks, security in sensor and vehicular networks, real-time systems, scheduling, real-time distributed systems. In the last 15 years, he served as program committee member of more than one hundred international conferences and workshops. He published several papers and books on real-time systems and communication networks.

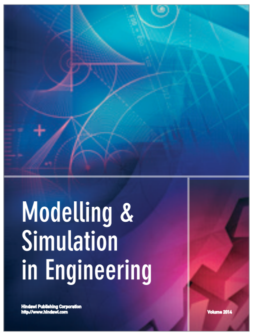
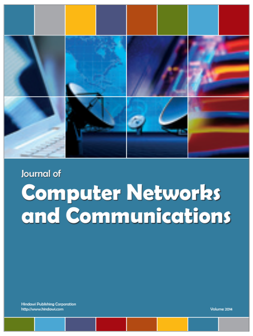
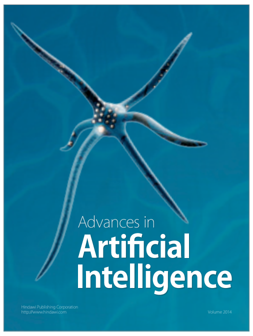
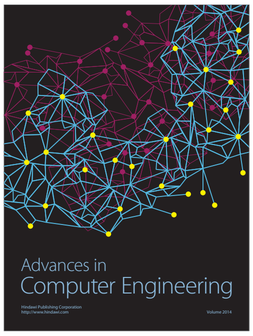
Franck Morvan received a PhD degree in Computer Science from Paul Sabatier University in 1994. He worked at the Dassault Data Services society. He is currently associate professor and member of the Institute of Research in Computer Science of Toulouse (IRIT). His main research interest are optimization in distibuted and parallel databases, mobile agents and mobile computing.

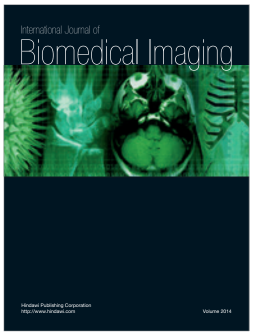
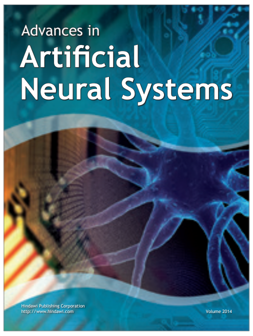
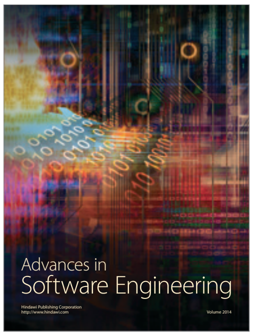
Abdelkader Hameurlain is full professor in Computer Science at Paul Sabatier University, Toulouse, France. He is a member of the Institute of Research in Computer Science of Toulouse (IRIT). His current research interests are in query optimization in parallel and large scale distributed environments, mobile databases, and database performance. Prof. Hameurlain has been the general chair of the International Conference on Database and Expert Systems Applications (DEXA'02). He was guest editor of two special issues of Internationa Journal of Computer Systems Science and Engineering on "Mobile Databases" and "Data Management in Grid and P2P Systems".

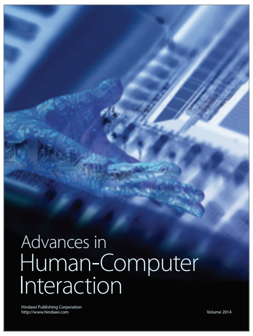
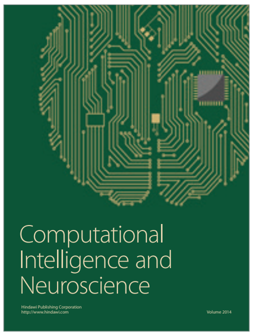
Nadem Marsit received the MS degree in computer science from University of Tunis (National Institut of Applied Science and Technologie). He received his Phd degree in computer science from Paul Sabatier University of Toulouse. He worked as a temporary teaching and research fellow (ATER) in IRIT laboratory and University of Toulouse 1. Actually he works as Software engineer in AIRBUS CIMPA on behalf of Ausy corp. His research interests include mobile computing, databases and real-time systems.


Advances in Multimedia

The Scientific World Journal

International Journal of Distributed Sensor Networks

Journal of Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in Fuzzy Systems


Modelling & Simulation in Engineering

Journal of Computer Networks and Communications

Advances in Artificial Intelligence

Advances in Computer Engineering

International Journal of Computer Games Technology

International Journal of Biomedical Imaging

Advances in Artificial Neural Systems

Advances in Software Engineering

Journal of Robotics

Advances in Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of Reconfigurable Computing

Journal of Electrical and Computer Engineering