The Impact of Filtering on Spatial Continuous Queries

Thomas Brinkhoff

Institute for Applied Photogrammetry and Geoinformatics (IAPG), FH Oldenburg/Ostfriesland/Wilhelmshaven (University of Applied Sciences), Ofener Str. 16/19, D-26121 Oldenburg, Germany, tbrinkhoff@acm.org

Abstract

Spatiotemporal database systems (STDBS) are primarily oriented to applications that track and present details about moving objects. Such applications must be kept informed about new, relocated, or removed objects that fulfil a given query condition. Consequently, a STDBS must inform its clients about these updates. Such queries are called *continuous queries*. The volume and frequency of transmissions is influenced by technical restrictions like the computing power of a client, the spatial distances a client is able to distinguish, and the maximum speed and throughput of the network connection. In this paper, filtering algorithms are presented that reduce the number of transmitted update operations. Two contradictory optimisation goals can be observed: First, to reduce the memory requirements of the STDBS for buffering these operations and, second, to reduce the volume and frequency of transmissions. Delaying or even not transmitting updates to a client may, however, decrease the quality of the query result. The impact of these algorithms is presented through discussion of a series of experiments.
Keywords: spatiotemporal database systems, moving objects, continuous queries

1 Introduction

Spatiotemporal database systems (STDBS) are an enabling technology for applications such as Geographic Information Systems (GIS), environmental information systems, and multimedia. In particular, the storage and retrieval of *moving objects* are central tasks of a STDBS. The investigation of spatiotemporal database systems is especially oriented to applications which are required to track and visualise moving objects. Many of these applications originate from the field of *traffic telematics.* This is a field in which techniques from the areas of telecommunication and computer science are combined and used in establishing traffic information and assistance services. Such applications require the management of moving objects, such as vehicles of a fleet (Wolfson *et al*., 1999) (Brinkhoff, 1999).

An important issue is the support of *mobile and location-based applications*. Mobile applications refer to locations and require the transmission of spatial or spatiotemporal data. The appearance of mobile applications has also had an impact on devices used for presenting data: Personal digital assistants (PDAs) or mobile telephones are used as clients. The computing power of such devices, however, is restricted compared to traditional computers. In addition, speed and throughput of wireless networks are subject to large variations.

The work presented in this paper is motivated by the two trends mentioned above. Applications tracking, and presenting moving objects require current and appropriate information about any new, relocated, or removed objects. Consequently, a STDBS must inform its clients about such update operations. The query causing this process is called *continuous query* (Terry *et al*., 1992). The 'result set' of a *spatial continuous query* is influenced by the update operations occurring in the database and by a given query condition. The query condition includes two aspects and consists of spatial predicates (e.g., a window query) and optionally of non-spatial predicates defining further selections (e.g., a selection of vehicles that are cars or motorbikes). For mobile applications, the processing capabilities of the client must also be taken into account.

Typical technical restrictions concern the computing power of the client, the spatial distances the client is able to (or wants to) distinguish, and the maximum speed and throughput of the connection between the STDBS and the client. Therefore, it is not advisable to transmit the complete result of a continuous query. Instead, a reasonable *filtering* must be performed. Two contradicting optimisation goals can be observed: First, to reduce the memory requirements of the STDBS for buffering the operations and, second, to reduce the volume and frequency of transmissions to the client. \Delaying or even not transmitting update operations to a client may, however, decrease the quality of the query result. Therefore, algorithms for filtering the result of a spatial continuous query are required to maintain a sufficient quality of the query result.

This paper starts with a short definition of the model used for describing moving objects and presents the main properties of spatial continuous queries. The second section introduces a first algorithm for processing continuous queries. More sophisticated algorithms are presented and analysed in the third section. They limit the memory requirements of the STDBS and reduce the volume and frequency of the transmissions. The quality of the query results and other properties of these algorithms are then experimentally investigated. Finally, the paper concludes with a summary and an outlook on future work.

2 Continuous Queries

2.1 Definitions

The following discussion assumes a STDBS, which stores the positions and other attributes of moving objects. In a temporal database, *valid time* and *transaction time* is distinguished. The valid time describes the time when a record is valid in the modelled reality. The transaction time is the time when a record is committed in the database system.

In the following, it is assumed that a moving object *obj* has an identifier *obj.id*. The object is described by a sequence of records obj_i ($i \in N$). Each record consists of a spatial location $obj_i.loc$ (short: loc_i), of a time stamp $obj_i.time$ (short: $time_i$) giving the beginning of the valid time, of a time stamp $obj_i.trTime$ (short: $trTime_i$) giving the transaction time, and of non-spatiotemporal attributes $obj_i.attr$ (short: $attr_i$). A *time stamp t* is represented by a natural number ($t \in N$). If the records obj_i and obj_{i+1} exist, the valid time of record obj_i corresponds to the interval $[time_i , time_{i+1})$. If no record obj_{i+1} exists for a record obj_i, the record obj_i is the current state of the corresponding moving object from the point of view of the STDBS. Furthermore, a final record obj_i may exist with $i > j$ for all records obj_j of this moving object. It indicates the end of the lifetime of the object. In order to simplify the discussion, we assume that $time_i \leq trTime_i$ holds.

We distinguish three *basic types of updates* concerning a moving object: 1, the insertion of a new object, 22, the modification of an existing object, and 33, the deletion of an existing object. With respect to the query condition of a distinct client, the type of an (modifying) update operation may change (Brinkhoff and Weitkämper, 2001). For example, the position of an object representing a vehicle has been modified in the database. If this vehicle leaves the query window of a client, this modification must be reclassified to a deletion for this particular client. Table 1 gives a summary of such reclassifications. The vehicle leaving the window is represented by the (yes, no) row and the modification column. Updates that do not need to be reclassified are shown as I1 and D1. While I2 and D2 denote reclassified updates.

Table 1. Reclassification of update operations

fulfils query condition?		original type of operation:		
previous record obj_{i-1}	current record obj_i	Insertion	deletion	modification
		reclassified type of operation:		
no	no	-	-	-
no	yes	insertion (I1)	. / .	insertion (I2)
yes	no	. / .	deletion (D1)	deletion (D2)
yes	yes	. / .	. / .	modification (M)

The reclassified type of an update operation may also determine *the interest of a client* in this operation: Deletions are typically of *high interest*. The same holds for insertions. For modifications, the situation may be different. In general, the number of modifications considerably exceeds the number of other operations. Therefore, it may be acceptable to skip some modifications, especially if the distance to the last reported position is small or the topology has not changed (e.g., the car is still on the motorway or is still in the same county). Then, the result set received by a client is not identical to the complete result set of a continuous query. In contrast to the assumptions done in queuing theory, not only a delaying but also a skipping of operations is acceptable. Another restriction concerns the database: the STDBS cannot store a reflection of all update operations each client has received as a result of the query condition – the cost would be prohibitive.

2.2 A First Algorithm

If a moving object is changed in the database, the STDBS will determine the affected clients and will reclassify the update operation type accordingly. If the update is of interest for a distinct client, the STDBS will call the procedure `collectUpdates` (see Fig. 1). In general, the new update operation is added to the set *client.ops*, which collects the operations intended for the client. *newOp* consists of the identifier of the corresponding object *(objId)*, of the current object representation *(curr)*, and the reclassified type of the operation (*opType)*. If an element concerning the same object already exists in the set *client.ops*, this element will be updated. As a result, and depending on the current type of operation, the element will be deleted or modified. `collectUpdates` guarantees that for each object at most one operation exists in the set *client.ops*.

The function `computeTransmission` determines the set of operations to be sent to a client. The STDBS calls this function before the updates are transmitted to a client. As a result, this first solution returns *client.ops* and then empties the set..

```
void collectUpdates (Client client, Operation newOp) {
// Adds an update operation newOp to a collection of a client.
 // case 1: the operation concerns no object referenced in the set
 if (newOp.objId ∉ {op.objId | op ∈ client.ops})
     client.ops = client.ops ∪ {newOp};
 // case 2: the operation concerns an object referenced in the set
 else {
     // determine the stored operation
     Operation oldOp = op∈client.ops with op.objId == newOp.objId;
     // if necessary, delete the operation from the set
     if ((oldOp.opType∈{I1,I2}) && (newOp.opType∈{D1,D2}))
         client.op = client.op \ {oldOp};
     // or update the type of operation and the description
     else if ((oldOp.opType∈{D1,D2}) && (newOp.opType∈{I1,I2})) {
         oldOp.opType = M;       // delete plus insert becomes modification
         oldOp.curr = newOp.curr;
     }
     // or update only the description
     else oldOp.curr = newOp.curr;
    }}

Set computeTransmission (Client client) {
// Determines the update operations to be sent to a client.
 Set sendOps = client.ops;
 client.ops = ∅;
 return sendOps;
}
```

Fig. 1. First version of the filtering algorithm

An aspect to consider is how best does one determine the point at which a set of updates should be transmitted to a client. One solution is to send a transmission as soon as the transaction time has exceeded a given period Δt. In this case, the size of set *client.ops* is only limited by the number of updates that a STDBS is able to process in the given period Δt. This number can be quite large. Assuming many clients use a STDBS in parallel, it may result in excessive memory requirements and poor scalability. Further, the performance of the client or the network connection to the client may restrict the number of operations that can be processed during a given period.

By reducing the period Δt, the first disadvantage may be reduced. However, the sum of transferred operations would increase. The reason for this increase is that the probability of replacing operations in the set *client.ops* decreases with shorter periods Δt. Time restrictions, that require a minimum period between two data transmissions, are another rationale against reducing Δt. The same argumentation will hold if the transmission is triggered by the size of the set *client.ops*. Only in the case of Δt=1, the algorithm computes the complete result of a continuous query. Otherwise, the transmission of operations may be delayed. By replacing outdated entries, the result set may be smaller than the complete result.

We can observe two contradictory optimisation goals: First, to reduce the memory requirements of the STDBS for buffering update operations and, second, to reduce the volume and frequency of transmissions to the client. In the following, we try to balance between these two objectives by modifying the initial algorithm.

3 Improving the Algorithm

According to (Brinkhoff and Weitkämper, 2001), table 2 summarises the parameters and functions, which can be used for describing the restrictions of a client. These parameters are used by the algorithms presented below.

Table 2. Parameters and functions used for describing the restrictions of a client

parameter	Description
maxOps	The maximum number of update operations that can be sent to a client by one transmission.
minOps	The minimum number of operations reasonable to be sent to a client by one transmission; it holds: minOps ≤ maxOps.
minPeriod	The minimum period between two transmissions to a client.
thr	A threshold for the measure of interest (see section 3.1).
function	**Description**
intr(ob_{prev},obj_{curr})	The measure of interest for operations that are not of high interest.
isRelevant(ob_{prev},obj_{curr})	Boolean function determining whether an update operation is relevant for a client or not.

3.1 Algorithm Observing the Restrictions of a Client

An algorithm, which determines the next update operations to be sent to a client for performing the continuous query, should observe the restrictions and measures described above. Like in section 2.2, the algorithm presented in Fig. 2 consists of the operations `collectUpdates` and `computeTransmission`.

The procedure `collectUpdates` is similar to the first version. A previous object description *(prev)* and an attribute *time* have been added to the elements of the set *client.ops*. The parameter *newOp* also includes an attribute *prev* representing the previous object representation in the database. A STDBS should be able to determine *newOp.prev* efficiently. For a new element in the set *client.ops*, the attribute *time* is generally set to the valid time of *newOp.prev*. An exception from this rule is the insert operation I1. Then, *time* is set to the valid time of the new object. If an element concerning the same object already exists in the set *client.ops*, this element will be updated. Note that the attribute *time* is not changed in this case. It still represents the time when the operation was inserted into *client.ops*.

```
void collectUpdates (Client client, Operation newOp) {
// Adds an update operation newOp to a collection of a client.
 // case 1: the operation concerns no object referenced in the set
 if (newOp.objId ∉ {op.objId | op ∈ client.ops}) {
     newOp.time = (newOp.opType == I1) ? newOp.curr.time : newOp.prev.time;
     client.ops = client.ops ∪ {newOp};
 }
 // case 2: the operation concerns an object referenced in the set,
 //         the attribute oldOp.time remains unchanged!
 else {
     // determine the operation
     Operation oldOp = op∈client.ops with op.objId == newOp.objId;
     // if necessary delete the operation from the set
     if ((oldOp.opType∈{I1,I2}) && (newOp.opType∈{D1,D2}))
         client.op = client.op \ {oldOp};
     // or update the type of operation and the description
     else if ((oldOp.opType∈{D1,D2}) && (newOp.opType∈{I1,I2})) {
         oldOp.opType = M;          // delete plus insert becomes modification
         oldOp.curr = newOp.curr;
     }
     // or update only the description
     else oldOp.curr = newOp.curr;
}  }

Set computeTransmission (Client client, Time currTime) {
// Determines the updates to be sent to a client. currTime: the current time
 // initialize the set of operations to be sent
 Set sendOps = ∅;
 // if the period is too short: return nothing
 if (currTime-client.timePrev) < client.minPeriod)
     return sendOps;
 // determine the operations of high interest
 Set o1 = {op∈client.ops | (op.opType)∈{I1,I2,D1,D2}) ∨
     (intr(op.prev,op.curr)≥client.thr)) ∧ isRelevant(op.prev,op.curr) };
 if ( |o1| > client.maxOps )
     sendOps = {op∈o1| client.maxOps elements having the oldest time stamps};
 else
     sendOps = o1;
 // determine further operations of interest
 if (|sendOps| < client.minOps) {
     Set o2 ={op∈client.ops∧op∉sendOps|isRelevant(op.prev,op.curr)};
     sendOps = sendOps ∪ {op∈o2| client.minOps-|o1| elements
                                  having the highest intr(op.prev,op.curr) };
 }
 // final actions
 if (sendOps ≠ ∅)
     client.prevTime = currTime;
 client.ops = client.ops \ sendOps;
 return sendOps;
}
```

Fig. 2. Filtering algorithm observing the restrictions of a client

The function `computeTransmission` has been completely modified: it determines the set of operations to be sent to the client observing the parameters and functions shown in table 2. First, the algorithm tests whether the time interval between the time, when update operations were sent to the client last, and the current time is sufficient. Then, a set of operations is determined. This set contains all operations of high interest and the operations whose measure of interest exceeds a given threshold *thr*. The elements are ranked according to the attribute *time*, which was determined by the operation `collectUpdates`. If necessary, this sequence will be cut by *maxOps*. If reasonable, further operations are added. The selected elements form the result of the function `computeTransmission`. These elements are removed from the set *client.ops*.

Note that non-relevant update operations are not removed from the set *ops*. There are two reasons for this. The first reason is to accumulate the movement. The STDBS cannot derive the last object representation obj_{last} sent to a client without explicitly storing previous information; the attribute *prev* of a new update operation *newOp* is often not identical to obj_{last}. Furthermore, keeping these operations in the set allows the value of the attribute *time to be preserved*. Otherwise, a sequence of non-relevant movements would repeatedly change the value of *time*. As a result, the ranking of this operation within other operations would stay on a low level. By keeping the first value of time, the ranking of the operation is improved over the duration of time. Fig. 3 illustrates these effects.

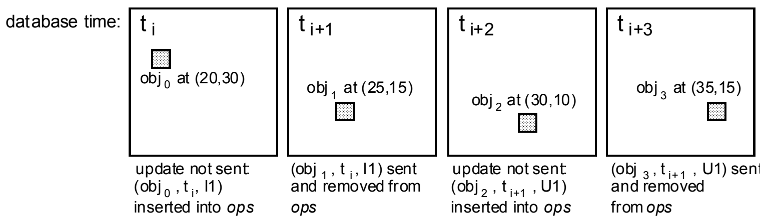

Fig. 3. Illustration of setting and keeping the attribute *time*

3.1.1 Discussion of the Time Complexity

The filtering algorithm depicted in Fig. 2 observes the parameters of table 2. However, its design has not considered any optimisations for reducing the time or the space complexity of the algorithm. Let us first discuss time complexity.

We can distinguish two different rankings using the attribute *op.time* ($op \in client.ops$) and the function *intr* computing the measure of interest. Therefore, we consider the two subsets ops_1 and ops_2 separately. ops_1 consists of all operations of high interest plus the relevant operations of lower interest. ops_2 consists of the other operations of lower interest. The assignment of an operation to one of these two subsets will only change, if the object description *op.curr* is changed. This is not problematic because in this case the algorithm `collectUpdates` is called, which can handle this case. The ordering of ops_1 is trouble-free because the attribute *op.time* will not be changed and the property *isRelevant* is static as long as the corresponding object *op.curr* is not changed. The same holds for the ordering of ops_2 because the function *intr* result does not change without changing *op.curr*.

The operations performed on the sets consists (a) of the insertion of elements, (b) of the search for an existing element (and its deletion) and (c) of the retrieval (and deletion) of the first *k* elements according to the ordering of the elements of

ops_1 and ops_2, respectively. To support these operations, the following options exist (n denotes the number of elements in *client.ops*):

- Operation (b) determines the existence of an operation in the set by using the identifier of the corresponding object. For an efficient search, we must know the value of the attribute, which defines the ordering: the valid time of the object representation originally inserted into *client.ops* and the object representation inserted before into *client.ops*. However, the original valid time is unknown for the calling STDBS. Therefore, the effort for performing operation (a) is *O(log(n))* and for operation (c) *O(maxOps*log(n))* if a balanced search tree according to the orders of ops_1 and ops_2 is used. However, the worst-case search time of (b) will be of *O(n)* in this case.
- Organising the sets by two redundant balanced search trees allows performing the operations (a) and (b) in *O(log(n))* and operation (c) in *O(maxOps*log(n))*. Each update operation, however, must be performed twice and as a result maintaining the two search trees increases the space requirements.

3.1.2 Discussion of the Space Complexity

The number of all current valid moving objects fulfilling the spatial and other non-spatial query conditions only limits the number of elements in the set client.ops. This number is denoted by N; it holds: $n \leq N$. In the worst case for each client performing the continuous query, the status of all moving objects fulfilling the query condition at the beginning or throughout the duration must be recorded in the set *client.ops.* Then, n will be quite large. Assuming a STDBS used by many clients in parallel, that means a huge memory and maintenance overhead. Therefore (and because of the time complexity), it is necessary to reduce this overhead by limiting the size of the set *client.ops*. This is the topic of the next section.

3.2 Restricting the Space Complexity

In an effort to restrict the memory demands the algorithm is modified by using a parameter, *maxSize*. The minimum value of *maxSize* is determined by the parameter *maxOps*: $maxSize \geq maxOps$. However, a higher value of *maxSize* would improve the results of the continuous query. If we restrict the size of the set, we will need a *replacement strategy*, which is required for a new important operation in the case that the set is full. The obvious replacement strategy is to remove the least important element from the set *client.ops* if the set has the size *maxSize*. One exception, however, must be observed. If we removed delete operations, this would have drastic impacts for the client: the client in most cases would never remove the corresponding object. Therefore, we must not remove such operations from the set *client.ops*. Instead, we remove another operation having an older time stamp *op.time* from the set. In the case where only delete operations exist, however, this approach does not work. A solution to neglect the space restrictions or to disregard the time restriction *minPeriod*, should be considered. This implies that the data would be sent earlier and therefore more data would be sent to the client than originally expected.

Removing elements from the set of operations has a further impact: A client may receive update or delete operations concerning objects unknown for the client because it has not received any insert operation for this object before. Consequently, such an update must be executed as an insertion and the client can ignore such a deletion.

4 Experimental Investigation

The experiments presented in the following investigate the applicability of the algorithms. Especially, the impact of the technical restrictions and of limiting the memory on the quality of the query results should be examined.

4.1 Test Data and Queries

For generating suitable test data, the generator for spatiotemporal data presented in (Brinkhoff, 2000) was used. This generator allows moving objects to be computed and observes several rules effective in simulating typical traffic situations. In our case, a street network consisting of 6,065 edges was used, and can be downloaded from the web site referenced in (Brinkhoff, 2000).Six object classes were defined. The maximum distance done by an object was between 1/250 and 1/8000 of the sum of the x-extension and the y-extension of the data space. The probability of a move per time stamp was 25%. The query condition used for the queries in the following tests is quite simple: it selects all objects lying in a query window having a size of 10% of the data space.

4.2 The Tests and Their Results

The following experiments were performed through an an implementation of the continuous query programmed in Java using Oracle 8i. The continuous queries were started at time stamp 640 and stopped at time stamp 1280. At time stamp 640, 281 moving objects were within the query window and at the end 387 objects. During the query, the complete number of operations to be transmitted to a client was 56,712. The measure of interest *intr* was computed as follows:

(1) $intr(obj_{prev}, obj_{curr}) := (time_{curr} - time_{prev}) + w_{loc} * dist\ (time_{curr}, time_{prev})$

The factor w_{loc} scales the Euclidean distance such that the influence of time and space is equalised. The threshold *thr* is set on a value that would be exceeded for *dist* = 0, if the period between the two operations was larger than 4**minPeriod*.

The first test series investigate the results of the continuous query for different minimum periods *minPeriod*. Fig. 4 gives an overview of the results. The results of a *minPeriod* of 1 correspond to the results computed by the algorithm presented in section 2.2; the other results are computed using the algorithm of section 3.1. The *omitting degree* describes the quality of the query result according to definition of (Brinkhoff and Weitkämper, 2001). The smaller the omitting degree, the better the quality of the query results. The omitting degree consists of two components: *timeOD* describes the temporal quality and *distOD* the spatial quality. For the sake of brevity, the definition of the omitting degree has not been presented here.

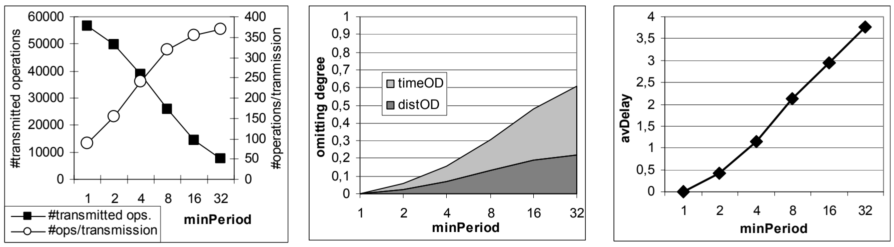

Fig. 4. Results depending on the minimum period *minPeriod*

The number of transmitted operations declines from 56,712 for a *minPeriod* of 1 to 7,769 for a *minPeriod* of 32, i.e. by a factor of 7.3. The number of operations per transmission increases by a factor of about 4.2. With increasing values of *minPeriod* the probability increases that an operation is updated by a new operation before it is transmitted to the client. As a result, the quality of the response decreases considerably; for a *minPeriod* of 32, we observe an omitting degree of about 0.61. The average delay is barely affected because it is only measured for transmitted operations and not for operations being replaced before sending them to the client.

In the next test series, the number of operations transmitted to the client *(maxOps)* was limited. The value of *minOps* was always set to *maxOps*/2. Fig. 5 shows the main results for a *minPeriod* of 4 and 8. In the unlimited case (unl), up to 292 and 362 operations per transmission occur.

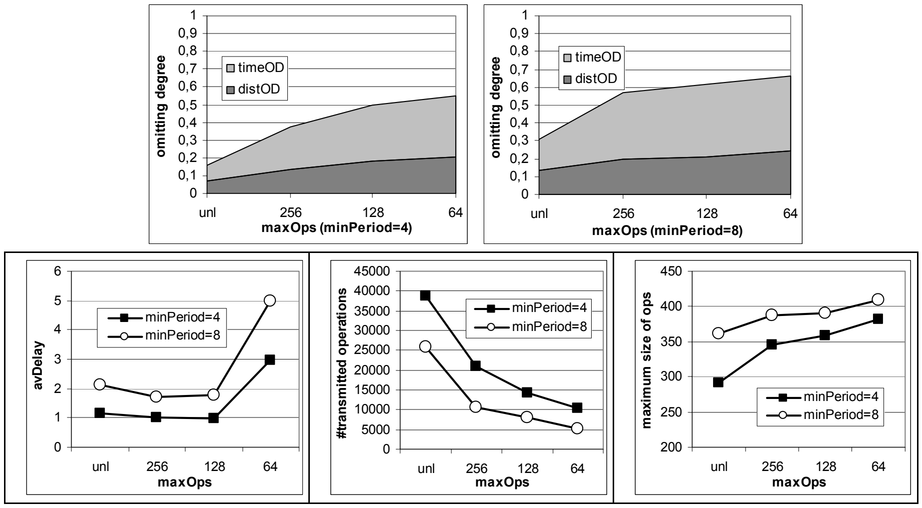

Fig. 5. Results depending on the number of operations per transmission

By limiting the number of operations, the total number of transmission operations also decreases: we observe factors of 3.8 and 4.9 between the unlimited case and a maximum number of 64 operations for a *minPeriod* of 4 and 8, respectively. Consequently, the measure *omitDeg* increases. However, the increase is relatively moderate, especially for the distance measure *distOD*. This observation demonstrates that the heuristics used by the algorithm for selecting the transmitted operations have success and compensate some of the loss of quality. The graphs depicting the average delay are quite interesting. They show that up to a certain point, the effect of limiting *maxOps* is that older operations in the set *client.ops* are replaced by newer operations. In this case, no impact on *avDelay* can be observed. Beyond this point, the transmission of operations is really delayed and *avDelay* increases. Another observation concerns the size of *client.ops*: the smaller *maxOps*, the larger the maximum size of this set. Therefore, the maximum size of *client.ops (maxSize)* is limited in the test series of Fig. 6. That means we investigate a version of the algorithm of section 3.1, which observes the space restrictions presented in section 3.2.

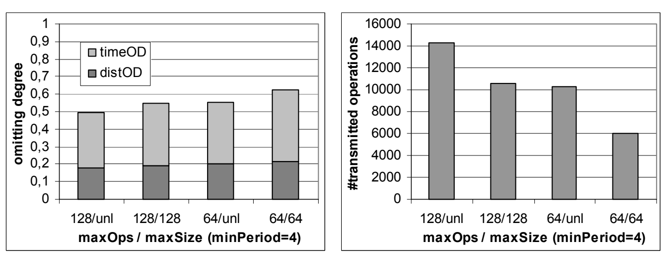

Fig. 6. Results depending on *maxOps* and *maxSize* (unl = unlimited)

Limiting *maxSize* leads to a further decline of the number of transmitted operations. Again, the impact on the quality of the query result is rather moderate. Especially, the distance portion of the omitting degree is almost unaffected.

We can summarise that the experiments have shown that the proposed algorithm allows limiting the number of transmitted operations as well as the number of operations buffered by the STDBS without a huge loss of quality.

5 Conclusions

In this paper, filtering algorithms for processing spatial continuous queries in a spatiotemporal database system (STDBS) have been presented. After presenting a first algorithm, we have observed two contradicting optimisation goals: First, to reduce the memory requirements of the STDBS and second, to reduce the volume and frequency of transmissions to the clients. To balance between these two objectives, an algorithm has been presented that observes different parameters modelling technical restrictions as well as the interest of a client in a distinct update operation. A restriction of the memory requirements of the algorithm has been achieved by using an adapted replacement strategy.

Delaying and not transmitting update operations to a client, however, decreases the quality of the query result. In an experimental investigation, the proposed algorithms have been examined measuring the quality of the query results. These tests have shown that the algorithm, which was finally proposed, allows limiting the number of transmitted operations as well as of the number of the operations buffered by the STDBS without a huge loss of quality. In particular, the impact on the distance between the locations of the transmitted object descriptions is rather moderate.

The definition of continuous queries in this paper was based on a simple model of moving objects. Therefore, future work should cover a definition using a more expressive data model. The same holds for the application. More complex is, for example, the detection of collisions for 3D moving objects (Mirtich, 2000). The experimental investigations presented in this paper have been based on a standard database system. A major drawback of using such a database system is the considerable effort necessary for determining the previous object description of an updated object. This disadvantage must be eliminated by extending the database system by a suitable buffering technique or by using (prototypes of) spatiotemporal database systems. More detailed performance investigations could then include the measurement of the processing time for performing continuous queries.. Another aspect to consider is the behaviour of the restricting parameters. In this paper, it is assumed that they do not change over time with respect to a client. However, the resolution of a client may be changed by performing a zoom operation. The parameters *minOps, maxOps* and *minPeriod* may be affected by the traffic of other network users or by a changed capacity of the connection (e.g., using the new mobile telephone standard UMTS, the maximum speed of a connection will depend on the distance of the mobile telephone to the next base station.) Therefore, efficient filter algorithms are required, which observe varying restrictions.

References

Brinkhoff T (1999) Requirements of Traffic Telematics to Spatial Databases. In: Proceedings 6th International Symposium on Large Spatial Databases. Hong Kong, China. Lecture Notes in Computer Science, Vol.1651, Springer, pp 365-369

Brinkhoff T (2000) Generating Network-Based Moving Objects. In: Proceedings 12th International Conference on Scientific and Statistical Database Management, Berlin, Germany, pp 253-255 (extended version accepted for GeoInformatica, Kluver)

Brinkhoff T, Weitkämper J (2001) Continuous Queries within an Architecture for Querying XML-Represented Moving Objects. In: Proceedings 7th International Symposium on Spatial and Temporal Databases, Redondo Beach, CA. Lecture Notes in Computer Science, Vol.2121, Springer, pp 136-154

Mirtich B (2000) Timewarp Rigid Body Simulation. In: Proceedings ACM SIGGRAPH 27th International Conference on Computer Graphics and Interactive Techniques, New Orleans, LA, pp 193-200

Terry D, Goldberg D, Nichols D, Oki B (1992) Continuous Queries over Append-Only Databases. In: Proceedings ACM SIGMOD International Conference on Management of Data. San Diego, CA, pp 321-330

Wolfson O, Sistla AP, Chamberlain S, Yesha Y (1999) Updating and Querying Databases that Track Mobile Units. Distributed and Parallel Databases 7(3):257-287