C3: Concurrency Control on Continuous Queries over Moving Objects

Jing Dai, Chang-Tien Lu
Virginia Polytechnic Institute and State University
7054 Haycock Road, Falls Church, VA 22043
{daij, ctlu}@vt.edu

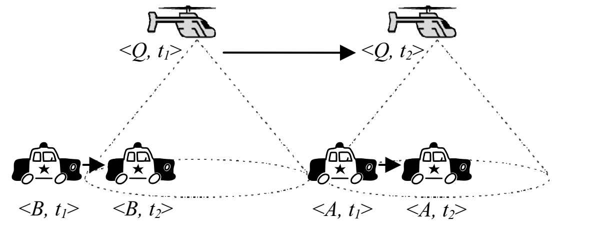
Abstract - **Moving object management approaches, especially continuous query processing techniques, have attracted significant research effort due to the broad usage of location-aware devices. However, little attention has been given to designing concurrency control protocols for continuous query processing. Existing concurrency control protocols for spatial indices are based on a single indexing tree, while popular continuous query processing approaches require multiple indices. In addition, continuous monitoring combined with frequent location updates challenges the development of serializable isolation for concurrent index operations. This paper proposes an efficient concurrent continuous query processing approach C3, which fuses scalable continuous query processing methods with lazy update techniques on R-trees. The proposed concurrency control protocol, equipped with intra- and inter-index protection, assures serializable isolation, consistency, and deadlock-freedom. The correctness of the proposed protocol is theoretically proven, and the experiment results demonstrated its scalability and efficiency.**

I. INTRODUCTION

The broad usage of location-aware devices, such as GPS and RFID, has promoted the applications of moving object management. The moving objects in the real-world need to be modeled and organized to efficiently process different queries. Examples of such applications include vehicle monitoring [1], [2] and flight tracking systems [3], [4] that manage real-time moving objects. To efficiently support these emerging applications, several spatial/spatial-temporal access methods have recently been proposed by accelerating frequent updates with hashing [5] and lazy update techniques [6]-[8]. In addition, continuous queries on moving objects have attracted significant research efforts due to their potential applications and the corresponding requirements on efficient data management. An example of continuous queries could be "tracking all ambulances within two miles to each patrol vehicle." To monitor a particular area of interest, it is inefficient to continuously reissue these range queries while the locations of ambulances and patrol vehicles keep changing. Several solutions have been proposed to support efficient continuous query processing via indexing both objects and queries [9]-[11].

To apply frequent location update and continuous query processing techniques in large scale multi-user systems, specific concurrency control protocols have to be designed to ensure the consistency of the database and the validity of query results. As stated in the Lowell Report [12], "We face major changes in the traditional DBMS areas, such as ..., concurrency control, ..., technology keeps changing the rules. These changing ratios require us to reassess storage management and query processing algorithms." Continuous query on moving objects in multi-user environments raises the following concurrency challenges: (1) Conflict from frequent movement: Frequent location updates and searches can cause conflicts when accessing spatial indices, and consequently can lead to inconsistent results. (2) Conflict by continuous monitoring: The conflicts could become even more serious while processing continuous queries, because both the objects and queries have to be monitored to refresh the results. (3) Inconsistency among indices: As scalable continuous query processing requires multiple indices, not only the consistency within an index, but also among these indices, has to be assured. Otherwise, the database may either miss queries or objects, or return incorrect results.

Existing concurrency control protocols only protect fundamental operations on a single indexing tree. Fig. 1 provides an example of inconsistent query results using these protocols. In this example, a patrol helicopter Q keeps tracking police vehicles within a given range of 0.5 mile. t_1 and t_2 are two consecutive query report timestamps. A and B are two police vehicles 1 mile away from each other, driving in the same direction as the helicopter. We assume that all location updates are submitted on time, and the query results are retrieved every time after the location updates at that timestamp have been submitted. Based on the existing spatial concurrency control protocols, a location update in this system consists of three atomic sub-operations: delete an old location, insert a new location, and refresh query results. With random execution orders of these sub-operations, location updates and query reports may exhibit inconsistent status [13]. Possible query result sets of Q at t_2 include $\emptyset$, $\{A\}$, $\{B\}$, or $\{A,B\}$, within which only $\{A\}$ is correct. Further details are discussed in Section III.B. Without proper serializable isolation, the above inconsistent scenarios may occur and thus cause serious consequences in critical applications, like flight control and battlefield information systems, where the relative locations of objects and queries are important. All these inconsistent scenarios can be avoided by a well designed concurrency control protocol for serializable continuous query processing.



Correct result for Q at t_2: ***A***
Possible results if without concurrency control:
Pseudo disappearance: on *A* or *A&B* → Ø, on *B* → *{A}*;
Back order: on *A* → Ø, on *B* → *{A,B}*, on *A&B* → *{B}*;
Pre-order: on *A* or *B* or *A&B*→*{A}*.

Fig. 1 Inconsistency of Continuous Query with Atomic Sub-operations.

The existing concurrency control protocols for spatial indices primarily consider fundamental operations (including searches, inserts, and deletes) on a single index. Record-oriented transaction management approaches [14] can protect complex operations. However, they are inefficient for processing continuous queries because they either lock resources until commit (2-phase locking) or result in a large number of roll-backs (version-based approaches) in the moving object scenario. Therefore, the lack of efficient concurrency control protocols for continuous queries limits the applicability of the moving object management systems in the real world.

This work proposes a Concurrency Control protocol for Continuous queries (C3) based on an efficient spatial access framework for continuous query processing. To the best of our knowledge, this is the first approach that applies lazy update techniques on scalable continuous query processing, and provides efficient serializable isolation on operations involving both on-disk and in-memory indices. Experiments demonstrated 160 ~ 380% performance gain from different lazy update buffers. Furthermore, the proposed concurrency control protocol exhibited up to 78% improvement when compared with existing index concurrency control integrated with record-oriented protocols. The major contributions of this paper are as follows:

- An efficient continuous query processing approach with lazy update techniques for moving objects is proposed based on R-trees.
- A sophisticated concurrency control protocol C3 is designed to assure serializable isolation, consistency, and deadlock-freedom for moving object indices.
- The correctness of the proposed concurrent operations is formally proven by analyzing their locking sequences and durations.
- The scalability and efficiency of the proposed framework were validated by a set of extensive experiments on benchmark datasets.

The rest of the paper is organized as follows: Section II reviews the related work on concurrency control protocols and moving object management. Section III illustrates the application scenarios for the proposed approach, and introduces the indexing structure and concurrency control protocol in this framework. The detailed algorithms for the concurrent operations are designed in Section IV. The correctness is analyzed in Section V. Section VI evaluates the performance of the proposed approach on benchmark datasets. Finally, Section VII concludes our work and suggests future directions.

II. Related Work

This section summarizes representative research achievements on the concurrency control on R-trees, frequent update for R-trees, and continuous query processing.

As one of the most popular multi-dimensional indexing structures, the R-tree [14] provides a robust tradeoff between efficiency and implementation complexity. Variants of the R-tree [15], [16] have been designed to optimize the indexing structure. To make the R-tree family applicable to real world systems, concurrency control protocols have been proposed to resolve the inconsistency in multi-user environments. The lock-coupling based algorithms [17], [18] release the lock on the current node only when the next node to be visited has been locked while processing search operations. To prevent the phantom update on the R-tree, the dynamic granular locking (DGL) has been proposed [19], where the empty space in any tree node can be locked as an external granule. For concurrent operations in read-dominant applications, GLIP [20] has been proposed to provide phantom protection on the R+-tree and its variants.

R-trees are usually considered as costly for updating, which makes them unsuitable for processing moving objects. Techniques utilizing hashing and lazy update have been designed to reduce the update cost of the R-tree and its variants. Table I lists several approaches for efficient update on R-trees and the corresponding techniques applied. The Frequent Update R-tree (FUR-tree) [5] processes delete operations directly from leaf nodes and simplifies insert operations if the location change is small. Lazy update approaches utilize buffer memory to reduce the I/O cost. The R-tree with update memos, RUM-tree [8], applies main memory buffer to cache delete operations, so that they can be processed later when the particular leaves are accessed. Lazy group update on R-tree, LGUR-tree [7], caches not only delete operations, but also insert operations. Another approach, the R^R-tree, constructs a memory-based buffer tree in addition to the disk-based R-tree to perform the lazy group update operations [6].

TABLE I. TECHNIQUES FOR EFFICIENT R-TREE UPDATE.

	FUR-tree [5]	RUM-tree [8]	LGUR-tree [7]	R^R-tree [6]
Leaf node Hashing	√			
Operation Buffer		√	√	
In-memory Tree				√

Continuous query is a common type of query that keeps monitoring moving objects in a certain area. One of the most challenging tasks in continuous query processing is to answer moving queries over moving objects. Several approaches have been proposed to tackle this problem by indexing both objects and queries. SINA [9] applies hashing techniques to join in-memory moving objects and queries, and performs further joins with on-disk objects and queries. SINA processes location updates in batches for optimal I/O costs. Another approach, MAI [10], constructs motion-sensitive indices for objects and queries, so that the update frequency can be reduced and prediction queries can be supported. A generic framework for continuous queries on moving objects [11] has been proposed to optimize communication and query re-evaluation due to frequent location updates.

Existing concurrency control protocols for the R-tree [19], [20] are neither sufficient to protect scalable continuous queries, nor suitable to handle the R-trees with lazy update buffers. The former requires protection on two independent indices, whereas the latter needs to assure the consistency on both in-memory and on-disk indices. It is not a trivial task to fuse concurrency control, continuous query, and lazy update techniques into a real-world moving object management system. Concurrent continuous query processing on moving objects has been proposed on a B-tree-based framework [13]. But it does not consider the operation protection over buffers. One potential solution is to adopt record-oriented transaction management techniques such as 2-phase locking (2PL) or versioning approaches [21] on indices. However, the 2PL strategy tends to lock the resources until the commit point, which performs similarly to the sequential execution on indices. The versioning approach requires a large number of different versions in frequent update scenarios, and thus leads to frequent undo/redo operations. The focus of this paper is to design an efficient concurrent continuous query processing approach on the R-tree-based access methods, such that frequent updates and continuous moving queries on moving objects are supported.

III. PRELIMINARIES

This paper provides a solution for concurrent continuous range queries on multi-dimensional moving object databases. In this environment, each range query keeps monitoring a spatial window and refreshing query results based on the object and query movement. Concurrent operations supported in this system include object location updates, query location updates, and query reports. An object location update operation inputs both the old and new locations of a spatial point, and updates the object database, object index, and query results. Similarly, a query location update operation inputs both the old and new positions of a spatial query window, and updates the query database, query index, and query results. A query report returns a set of data objects that currently overlap with a given query. These operations should not interfere with each other, and the outputs of query report operations should reflect the current consistent state of the database. Concurrent fundamental operations including insert, delete, and search can be inferred from the continuous query processing in C3.

To achieve scalability, three indices are utilized in this design, for moving objects, moving continuous queries, and query results, respectively. The index for query results is the join of the indices of objects and queries, and is consistent with the updates on these two indices. The consistency of these indices is assured by a concurrency control protocol.

The proposed C3 works for the R-tree-based spatial access methods with lazy updates. This access method comprises the features of both the lazy group update and update memo techniques, so the proposed protocol can work on those R-trees with any of these techniques for both continuous and snapshot queries. In order to focus on the concurrency control protocol, the access method is generalized by only maintaining the current locations of queries and objects. However, it is convenient to extend the concurrency control protocol to velocity-sensitive indices, such as MAI [10]. Furthermore, C3 can be generalized to a basic model that consists of three indices, two independent indices and the third as the joint of the first two. The updates on either one of the independent indices will be reflected in the joint results. The proposed protocol can be applied on this generalized structure for serializable isolation.

To specify the problem to be solved, several assumptions for the application environment are made:

- **Point objects**: Moving objects are represented by spatial points; each object reports its new location to the database during movement.
- **Window queries**: Moving queries are represented by their query windows (spatial boxes); each query reports its new query window to the database during movement.
- **Lock manager**: There exists a lock manager to support different lock types and to maintain locks. It has a system counter to assign a unique timestamp to each operation.

In addition, we assume that the operations submitted to the database are processed without timeout restriction. The above assumptions are practical in real-world applications. Some previous work, such as SINA [9], adopted a different approach that handles location updates in batches. With our assumptions, new locations are updated immediately after being reported. This work aims to assure the continuous consistency between query results and movements; the relative positions of items are important in many applications where concurrency control should be applied. For these applications, our approach has the advantage of handling updates without losing movement details (e.g., missing trajectories of fast-moving objects/queries, returning incorrect results due to aligned update time). Meanwhile, using the generalized 3-index model, the proposed concurrency control protocol can be integrated into a system like SINA for concurrent operations in batches. Based on these assumptions, the design of access methods and the corresponding concurrency control protocol are introduced in the following subsections.

A. Access Framework

The proposed concurrency control protocol is based on a generalized spatial access framework that integrates the existing techniques for frequent update and continuous query processing. As summarized in Table II, the generalized access method applies two R-trees with lazy group update for insert operations and with update memo for delete operations. One R-tree is constructed for indexing moving objects (*O-tree*, shown in Fig. 2) and the other is for indexing moving queries (*Q-tree*, shown in Fig. 3). The construction methods of *O-tree* and *Q-tree* are exactly the same. In addition to *O-tree* and *Q-tree*, there is a hash-based array, *Q-result* (shown in Fig. 4), to store all the results for continuous range queries.

On both *O-tree* and *Q-tree*, the lazy group update requires one insert buffer *I-buffer* (dashed boxes connected to each non-leaf node in Fig. 2) for each non-leaf node. *I-buffer*s temporally store inserted objects or queries on an appropriate level, and if full, push the largest group of inserted objects or queries down to the particular *I-buffer* on the next level or to the leaf node [7]. Each entry of an *I-buffer* has the form of (*Oid/Qid*, *MBR*, *target_child*, *timestamp*).

TABLE II. COMPONENTS IN INDEXING STRUCTURE.

Component	O-tree	Q-tree	Q-result
Function	Index moving objects	Index continuous queries	Store continuous query results
Implementation techniques	I-buffer to cache insertion; D-buffer to cache deletion	I-buffer to cache insertion; D-buffer to cache deletion	Hash array

On the other hand, efficient updates need a delete buffer *D-buffer* (dashed box beside the tree in Fig. 2) for each R-tree. *D-buffer*s cache the delete operations by recording the object/query IDs, the number of obsolete records for that ID, and their latest timestamps, and remove the obsolete records in leaf nodes when processing garbage collection [8]. Each entry of a *D-buffer* has the form of (*Oid/Qid*, *#_obsolete*, *timestamp*).

On either *O-tree* or *Q-tree*, a range search needs to traverse the R-tree with *I-buffer* to locate records overlapped with the search range. In this structure, search results can appear not only in leaf nodes, but also in *I-buffer*s. Before outputting the objects, the *D-buffer* has to be checked to remove obsolete objects from the results. An insert operation on either index tree first tries to insert a given item into the *I-buffer* associated with the root node of the R-tree. If the target *I-buffer* is full, it will be re-organized by: 1) removing obsolete items by checking the *D-buffer*; 2) executing lazy group update to push items into an *I-buffer* on the next level, whose associated node is chosen to include this item based on the R-tree insertion algorithm, or into a leaf node if it reaches the leaf level of the R-tree. A delete operation on either of these indexing trees only needs to add this delete record to the *D-buffer* with the current timestamp. A *D-buffer* can be cleaned by visiting the leaf nodes to remove obsolete items. The sizes of the *I-buffers* and *D-buffers* are much smaller than that of the trees. The impact of buffer size on location updates has been studied in [7], [8].

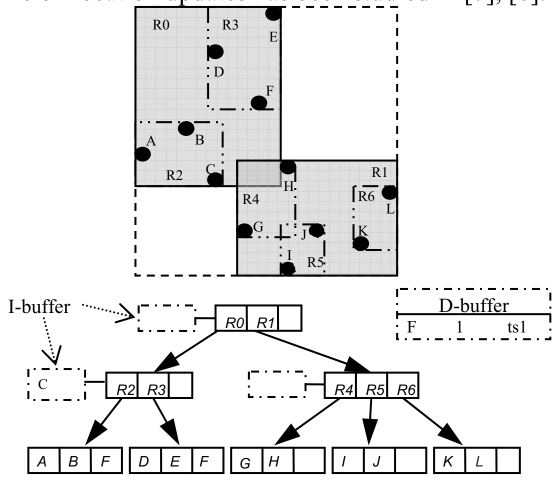


Fig. 2 An Example of *O-tree* with *I-buffer*s and *D-buffer*.

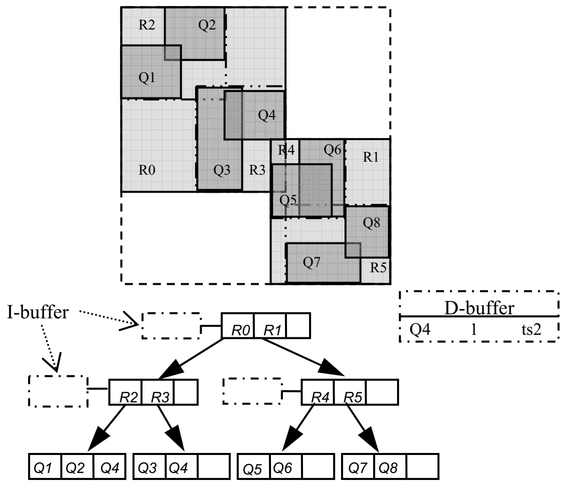


Fig. 3 An Example of *Q-tree* with *I-buffer*s and *D-buffer*.

Q-result (shown in Fig. 4) is a hash-based array to store all the results for continuous range queries. It is hashed by query IDs, and each particular entry corresponds to a continuous query. Each entry of *Q-result* is in the form of (*Qid*, *obj_list*), corresponding to a query ID and the list of objects covered by the query. The *obj_list* also contains the timestamp of each object in the list. The instance of *Q-result* in Fig. 4 reflects the data and query sets in Fig. 2 and Fig. 3 accordingly. For example, the query *Q2* covers the object *D*, therefore the entry *Q2* in the *Q-result* contains *D*. A query report on *Q2* can directly retrieve *D* from the *Q-result* without accessing the indexing trees.

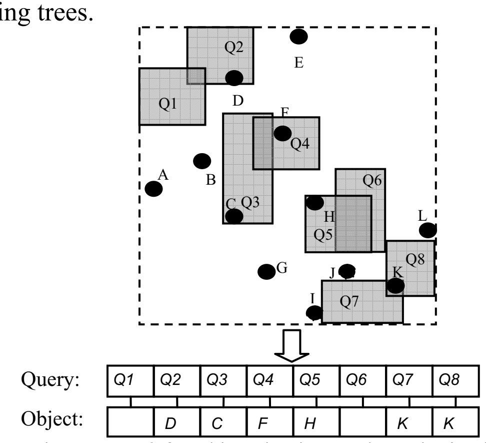


Fig. 4 *Q-result* for Objects in Fig. 2 and Queries in Fig. 3.

The continuous queries on this indexing framework are processed via three operations, query report, object location update, and query location update. The details of these operations are discussed in Section IV.

B. Concurrency Control Protocol

Continuous query processing requires an appropriate concurrency control protocol to ensure valid results. Take the scenario in Fig. 1 as an example, inconsistent results are caused by unserializable processing schedules. Suppose each movement contains three atomic phases supported by existing protocols: *D* for the deletion of an old location, *I* for the insertion of a new location, and *R* for refreshing query results. And the atomicity of each single phase is assured by an appropriate concurrency control protocol in place. In addition, let qR_{t2} denote the query report for Q at t_2, which is also atomic.

The situation that returns empty set at t_2 is called **pseudo disappearance**, because the vehicle A seems disappeared in Q during its movement. This happens when a processing sequence contains $\ldots\rightarrow A.D_{t2}\rightarrow Q.R_{t2}\rightarrow A.I_{t2}\rightarrow qR_{t2}\rightarrow A.R_{t2}\rightarrow\ldots$, where A has been deleted from the database when Q updates its results, and there are no more updates occur before the query report for t_2. The vehicle B will be returned at time t_2 in a scenario called **back order**, where the query seems staying at its previous position while some objects have already updated their locations. This occurs when a processing sequence contains $\ldots\rightarrow B.R_{t2}\rightarrow Q.D_{t2}\rightarrow qR_{t2}\rightarrow Q.I_{t2}\rightarrow\ldots$. In this case, B updates its location and adds itself to the result set of Q before Q's location is updated, and the query report is processed before Q re-evaluates its results. Back order may also result in an output $\{A,B\}$ at t_2, where only B is back ordered and A is in normal status. Such a processing sequence may contain $\ldots B.R_{t2}\rightarrow Q.I_{t2}\rightarrow A.R_{t2}\rightarrow qR_{t2}\rightarrow Q.R_{t2}\ldots$. In contrast to back order, another scenario is named **pre-order**, in which the queries are updated while some location updates for objects are delayed. In this example, pre-order on A, which may contain a processing sequence like $\ldots Q.R_{t2}\rightarrow A.D_{t2}\rightarrow qR_{t2}\ldots$, will still output $\{A\}$ as the result of Q at t_2, because in this situation, Q evaluates and outputs its results before the new location of A is updated, and both $<A,t_1>$ and $<A,t_2>$ intersect with $<Q,t_2>$. All these inconsistent processing sequences have to be prevented by the concurrency control protocol.

In this protocol, the lockable items include leaf nodes and the external granules of nodes (defined in DGL [19]) on both trees, entries in each *D-buffer*, and entries in *Q-result*. *I-buffer*s in both trees do not need specific locks, because they are attached to certain internal nodes which can be locked by the external granule. Following the convention of lock-coupling approaches [19], the types of locks that are utilized in the proposed protocol include *S* (shared lock), *X* (exclusive lock), *IX* (Intention to set *X* locks), *IS* (Intention to set *S* locks), and *SIX* (Union of *S* and *IX* lock).

In the proposed protocol, lock requests can be conditional or unconditional for different purposes. A conditional lock request means if the lock cannot be granted immediately, the requester will cede the lock request. On the contrary, an unconditional lock request means that the requester is willing to wait until the lock can be granted. In the proposed concurrent framework, most lock requests are unconditional. Only a small portion of lock requests are designed as conditional to prevent unnecessary process and avoid deadlocks.

In summary, the proposed concurrency control protocol supports serializable isolation by providing protection from the following issues: 1) inconsistency within each indexing structure, 2) inconsistency among *D-buffer*s, *O-tree*, *Q-tree*, and *Q-result,* and 3) deadlock caused by accessing multiple indices.

IV. Concurrent Operations

The proposed concurrency control protocol supports concurrent operations for continuous query and moving object management, including query report, object location update, and continuous query location update. These operations can be simultaneously processed without interfering with each other. These concurrent operations are described in the following subsections.

A. *Query Report*

The query report operation retrieves the moving objects covered by a continuous query. This operation takes a query ID as input and returns a set of object IDs. In the proposed indexing framework, it reads the particular entry in the *Q-result* and validates the results by looking up the *D-buffer* of *O-tree*, because the *Q-result* is a hashed array that may contain obsolete objects. In detail, with C3, the query report first places an *S-lock* on the entry with a given query ID in the *Q-result*, and reads the corresponding *obj_list*. After that, it requests *S-locks* on the *D-buffer* of *O-tree* for all the objects that contained in *obj_list*, and then removes obsolete objects from *obj_list*. At last, this operation returns the remaining objects and releases all the *S-locks* it has requested.

For example, based on the *Q-result* in Fig. 4, the query report with the input *Q5* requests an *S-lock* on the entry *Q5* in the *Q-result* and then finds the object *H* in the entry. An *S-lock* is then requested on the entry of *H* in the *D_buffer*. After validating *H*, the algorithm releases all these *S-locks* and returns *H* as the final result. The algorithm of the query report is illustrated in Algorithm 1. This operation can be requested by clients or triggered by the corresponding *Q-result* updates.

```
Algorithm Query_Report
Input: Qid: Query ID
Output: S: Set of Objects
S-lock(Q-result.entry[Qid]);
S = Q-result.entry[Qid].obj_list;
For each pair (Oid, ts) in S
        S-lock(O-tree.D-buffer.entry[Oid]);
        If Oid in O-tree.D-buffer
                If O-tree.D-buffer.entry[Oid].ts > ts
                        S = S – (Oid, ts);
        Unlock(O-tree.D-buffer.entry[Oid]);
Unlock Q-result.entry[Qid];
Return S;
```

Algorithm 1. Query Report.

B. *Object Location Update*

The object location update operation updates the location of an object, as well as the results of affected queries. It takes the new location of an object and performs a lazy update on the *O-tree* and an update on the *Q-result*. There are three phases in this operation, location update on the *O-tree*, point search on the *Q-tree*, and update on the *Q-result*, as shown in Algorithm 2. The details of these phases are presented as follows.

```
Algorithm Object_Location_Update
Input: Oid: Object ID, loc_old: Old Location of Object, loc_new: New Location of
Object, O-tree: Index Tree of Objects, Q-tree: Index Tree of Queries, Q-result: Result
Sets for Queries
Output: Nil
ts = get_timestamp();
QList = Nil; //set of queries that cover the point
//Phase 1. O-tree location update
//delete old location
X-lock(O-tree. D-buffer.entry[Oid]; //avoid operations on same object
O-tree.D-buffer.Add(Oid, #_obsolete, ts);
//insert new location
X-lock(O-tree.root.ext);
curNode = O-tree.root;
Add (Oid, loc_new, ts) to curNode.I-buffer;
Enlarge curNode.MBR to cover loc_new;
While curNode.I-buffer is full and not curNode.isLeaf
        nextNode = curNode.I-buffer.childForUpdate();
        X-lock(nextNode.ext);
        curNode.I-buffer.groupUpdate(); //push the largest group of contents to its
                                                        target child
        Unlock(curNode.ext);
        curNode = nextNode;
Unlock(curNode.ext);
//Phase 2. Q-tree point search
S-lock(Q-tree.root.ext);
curNode = Q-tree.root;
While not curNode.isLeaf
        QListTmp = curNode. I-buffer.find(loc_new);
        If (S-lock(Q-tree. D-buffer.entry[QListTmp], Conditional)) //conditional lock,
                                                        true if the lock is granted
            X-lock(Q-result.entry[QListTmp]); //avoid operations on same query
            QList.add(QListTmp);
        nextNode = curNode.findEntry(loc_new);
        S-lock(nextNode.ext);
        Unlock(curNode.ext);
        curNode = nextNode;
QListTmp = curNode.find(loc_new);
If(S-lock(Q-tree. D-buffer.entry[QListTmp], Conditional)) //conditional lock
        X-Lock(Q-result.entry[QListTmp]); //avoid operations on same query
        QList.add(QListTmp);
Unlock(curNode.ext);
QList = Q-tree. D-buffer.filter(QList); //remove obsolete queries in QList
Unlock(Q-tree. D-buffer.entry[QList]); //
//Phase 3. Q-result Update
For each Qid in QList
        Update Q-result.entry[Qid].obj_list by adding (Oid, ts);
        Unlock(Q-result.entry[Qid]);
UnLock(O-tree. D-buffer.entry[Oid]);
Return;
```

Algorithm 2. Object Location Update.

Phase 1 - *O-tree* location update: It updates the *O-tree* by inserting the new location and deleting the old location in a lazy manner. It first requests an *X-lock* on the corresponding object in the *D-buffer* of the *O-tree* to avoid conflict on accessing the same object. This *X-lock* will be kept until the end of this operation to avoid deadlock. After adding the delete record in the *D-buffer* of the *O-tree*, the algorithm performs a lazy group insertion on the *O-tree*, which attempts to insert the new location into a higher level *I-buffer*. The locking strategy for lazy group insertion is similar to the insertion in DGL, except that once the external of a tree node is locked, the *I-buffer* attached to it is also treated as locked.

Phase 2 - *Q-tree* point search: It queries the *Q-tree* using the new location of the object to find all the queries that cover this new location. The actual retrieval is performed on the corresponding *I-buffer*s and leaf nodes. A locking strategy similar to the search in DGL is applied on the indexing tree. Additionally, when a query is identified to cover the object, the corresponding entry in the *D-buffer* of the *Q-tree* will be *S-lock*ed and scanned to validate the query. Note that these *S-lock*s on the *D-buffer* of the *Q-tree* are unconditional, which means if any of these queries is *X-lock*ed by other operations, this object location update will cede that occupied query. This is because if a query is locked by a query location update, it will be re-evaluated based on its new location. So there is no need to include this query in this object location update. This unconditional lock can also prevent the deadlock between object location updates and query location updates. Once the affected queries are found, an *X-lock* will be requested on the corresponding entries in the *Q-result* to avoid conflict accesses on the same query. All the *S-lock*s on the *D-buffer* of the *Q-tree* are released by the end of phase 2 to allow accesses from other concurrent operations.

Phase 3 - *Q-result* update: It adds the object and the corresponding timestamp to the query results of all the queries that have been found in phase 2. Because these entries in the *Q-result* have been locked in phase 2, they can now be directly updated. The locks on the *Q-result* and the *D-buffer* of the *O-tree* are released at the end of this operation.

An example based on the objects and queries in Fig. 2 and Fig. 3 can demonstrate this object location update. Suppose the object *C* is moving into the region of node *R4* and also covered by the query *Q5*. The system first locks the entry of *C* in the *D-buffer* of the *O-tree*, although there is no record for *C* yet. Then an entry (*C, 1, ts*) is inserted into the *D-buffer*. *C* is then inserted into the *I-buffer* of the root node in the *O-tree* with timestamp *ts*. In phase 2, a point search is performed on the *Q-tree* using the new location of *C*, and *Q5* is retrieved. The entry of *Q5* in the *D-buffer* of the *Q-tree* is *S-lock*ed, and the entry of *Q5* in the *Q-result* is *X-lock*ed. After checking the *D-buffer*, *Q5* is confirmed as the affected query by *C*. The *obj_list* of *Q5* is now updated to contain *C* and *H*. Finally, all the locks are released.

C. Query Location Update

The query location update operation handles the location change of a query. This change could be on the location or the size of query window, or both. This operation takes the new search window of a given query as input, and updates the *Q-tree* and the *Q-result* accordingly. Similar to the object location update operation, this algorithm consists of three phases, namely, location update on the *Q-tree*, range search on the *O-tree*, and update on the *Q-result*. Timestamp *ts* is assigned at the beginning of the processing, so that the lazy update can have a sequential order for the update records.

```
Algorithm Query_Location_Update
Input: Qid: query ID, loc_new: new query window, O-tree: object index tree, Q-tree:
query index tree, Q-result: results of the queries
Output: Nil

ts = get_timestamp();
OList = Nil; //set of objects that are covered by the new query

//Phase 1. Q-tree location update
//delete old window
X-lock(Q-tree. D-buffer.entry[Qid]); //avoid operations on same query
Q-tree. D-buffer.Add(Qid, #_obsolete, ts);
//insert new window
X-lock(Q-tree.root.ext);
curNode = Q-tree.root;
Add (Qid, loc_new) to curNode.I-buffer;
Enlarge curNode.MBR if needed;
While curNode. I-buffer is full and not curNode.isLeaf
        nextNode = curNode. I-buffer.childForUpdate();
        X-lock(nextNode.ext);
        curNode. I-buffer.groupUpdate; //push the largest group of contents to its
                                          target child
        Unlock(curNode.ext);
        curNode = nextNode;
Unlock(curNode.ext);
X-Lock(Q-result.entry[Qid]); //avoid conflict from update on same query occur in
                                 middle
UnLock(Q-tree. D-buffer.entry[Qid]);

//Phase 2. O-tree range search
S-lock(O-tree.root.ext);
curNode = O-tree.root;
while not curNode.isLeaf
        OListTmp = curNode. I-buffer.find(loc_new);
        If(S-lock(O-tree. D-buffer.entry[OListTmp], Conditional)) //conditional lock
            OList.add(OListTmp);
        nextNode = curNode.findEntry(loc_new);
        S-lock(nextNode.ext);
        Unlock(curNode.ext);
        curNode = nextNode;
OListTmp = curNode.find(loc_new);
If(S-lock(O-tree. D-buffer.entry[OListTmp], Conditional)); //conditional lock
        OList.add(QListTmp);
Unlock(curNode.ext);
OList = O-tree. D-buffer.filter(OList);

//Phase 3. Q-result update
Q-result[Qid].OList=OList;
Unlock(Q-result.entry[Qid]);
Un-lock(O-tree. D-buffer.entry[OList]);
Return;
```

Algorithm 3. Query Location Update.

Phase 1 - *Q-tree* location update: It performs a lazy update on the *Q-tree*. It first requests an *X-lock* on the *D-buffer* entry of the *Q-tree* for that query, so that the conflict caused by accessing the same query can be avoided. After that, this operation adds the deletion record to the *D-buffer* of the *Q-tree*, and performs a lazy group insertion on the *Q-tree* with the new query window. The locking strategy applied for insertion on the *Q-tree* is similar to phase 1 in the object location update. By the end of phase 1, the corresponding entry in the *Q-result* is exclusively locked, and the *X-lock* on the *D-buffer* of the *Q-tree* is released, so that the particular query is always under protection.

Phase 2 - *O-tree* range search: It queries the new query window on the *O-tree* to retrieve all the objects that are covered. This range search scans the nodes and their *I-buffer*s on the traversal path, and requests *S-lock*s for the covered granules and the corresponding entries in the *D-buffer* of the *O-tree*.

Note that the *S-lock*s on the *D-buffer* of the *O-tree* are unconditional, which means if these objects are *X-lock*ed by other operations, this query location update will cede those objects. This is because if an object is locked by object location update, it will be added to the corresponding queries based on its new location. So there is no need to include this object in the *Q-result* update. This unconditional lock can also prevent the deadlock between object location update and query location update.

Phase 3 - *Q-result* update: It replaces the particular entry of the *Q-result* with the new set of objects that have been found as the results. The *X-lock* on the entry of the *Q-result* and the *S-lock*s on the corresponding entries of the *D-buffer* of the *O-tree* are released immediately after the update is completed. The details of the algorithm are shown in Algorithm 3.

The algorithm of query location update can be illustrated using an example based on Fig. 2 and Fig. 3. Assume the query *Q8* is moving upward within *R1* and covers the object *L* with its new window. It can still be included in the extended leaf node *R5*. The algorithm first locks the entry of *Q8* in the *D-buffer* of the *Q-tree*, although there is no record for *Q8* yet. Then an entry (*Q8, 1, ts*) is inserted into the *D-buffer* and *Q8* is inserted into the *I-buffer* of the root node in the *Q-tree*. An *X-lock* is requested on the entry for *Q8* in the *Q-result* before releasing the lock on the *D-buffer* of the *Q-tree*. In phase 2, a window search is performed on the *O-tree* using the new query window of *Q8*, and the object *L* is retrieved. The entry of *L* in the *D-buffer* of the *O-tree* is *S-lock*ed before checking its validity. After *L* is confirmed as an object covered by *Q8*, the *obj_list* of *Q8* is replaced by *L*. Finally, all the locks are released.

D. *Garbage Clean*

Garbage clean for the proposed framework consists of two procedures, *I-buffer* clean and *D-buffer* clean. An *I-buffer* clean is a straightforward process. It pushes the valid items in an overflowed *I-buffer* to the next level on the tree. The concurrency control protocol requests *X-lock*s on the external granules of the corresponding tree nodes involved in the *I-buffer* clean procedure.

A *D-buffer* clean procedure maintains the size of *D-buffer*s. It compares the timestamps of the entries in leaf nodes or *I-buffer*s with the corresponding entries in a *D-buffer*, and removes the obsolete items in leaf nodes/*I-buffer*s. Meanwhile, the corresponding deletion records in *D-buffer*s are updated. This can be triggered by updating a leaf node/*I-buffer* or moving a token. The proposed concurrency control protocol protects this *D-buffer* by requesting *X-lock*s on the involved leaf nodes/*I-buffers* and the items in the *D-buffer* before comparing their timestamps. If the item in a leaf node/*I-buffer* is obsolete, the operation deletes the entry in the leaf node/*I-buffer* and updates the entry in the *D-buffer*. After the clean process of that item is completed, both locks will be released.

V. Correctness

The proposed concurrency control protocol C3 assures serializable isolation, consistency, and deadlock-freedom on the generalized access framework. Serializable isolation means the results of any set of concurrent operations equal to those from the sequential processing of the same set of operations. Consistency refers to the feature that the results always reflect the current committed status. Deadlock-freedom means any combination of the concurrent operations does not cause any deadlock. The correctness of this concurrency control protocol is discussed as follows by analyzing the lock sequences and durations of each operation.

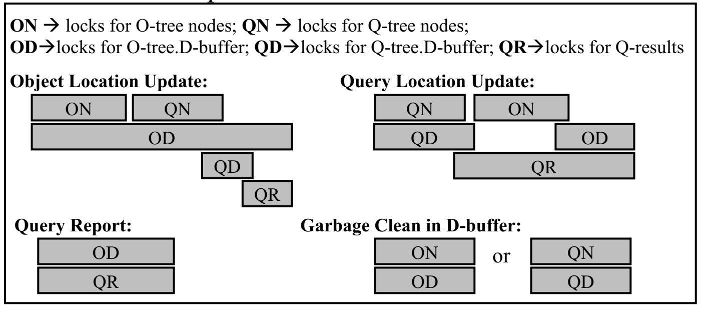

ON → locks for O-tree nodes; **QN** → locks for Q-tree nodes;
OD→locks for O-tree.D-buffer; **QD**→locks for Q-tree.D-buffer; **QR**→locks for Q-results
Object Location Update:
ON
QN
OD
QD
QR
Query Location Update:
QN
ON
QD
OD
QR
Query Report:
OD
QR
Garbage Clean in D-buffer:
ON
OD
or
QN
QD
Fig. 5 Lock Durations for Concurrent Operations.

Fig. 5 abstracts the order and duration of the locks requested in each operation, including object location update, query location update, query report, and garbage clean in *D-buffer*s. The garbage clean in *I-buffer*s only processes inside an R-tree, so it does not cause any correctness issue with inter-structure operations. The abbreviations in Fig. 5 indicate the locks on different structures. The items *ON* and *QN* are the locks inside the R-trees, while the items *OD*, *QD* and *QR* are the locks for the inter-structure protection. Objects and their corresponding *O-tree* nodes are locked in *ON* and *OD*, while queries and their corresponding *Q-tree* nodes are protected in *QN*, *QD* and *QR*. The horizontal span of each bar represents the time period that the lock is granted. Based on the algorithms, search operations request *S-lock*s, and update operations request *X-lock*s. The object location update and query location update will not request *S-lock*s on the same substructure. The query report only requests *S-lock*s, while garbage clean in *D-buffer*s places *X-lock*s. Among these locks, *ON* and *QN* are gradually requested by traversing the tree; the other locks for each bar are granted at almost the same time.

Serializable isolation: The proof of serializable isolation contains two parts, serializable isolation on the single tree and among the *O-tree*, *Q-tree*, and *Q-result*. The serializable isolation on a single R-tree has been proved [19]. A similar proof can show that *O-tree* and *Q-tree* are internally serializable, because the sub-operations on each single tree (*ON* and *QN*) are protected like on an R-tree, except that the locks on tree nodes cover the associated *I-buffer*s.

On the other hand, the serializable isolation among the *O-tree*, *Q-tree*, and *Q-result* can be proved based on the theory that a group of transactions are serializable if and only if their conflict graph has no cycles [22]. We prove this in the following lemma.

Lemma 1: Object location updates (*OLU*), query location updates (*QLU*), query reports, and garbage cleans are serializable given that any sub-operations involve a single indexing tree are serializable.

Proof: We prove this lemma using induction. Given that any sub-operations involve a single index tree are serializable, because of the conditional lock applied in the algorithm, the sub-operations on index tree nodes and *I-buffer*s are serializable to each other. Therefore, we only need to consider the sub-operations corresponding to *OD*, *QD*, and *QR*. Fig. 6 illustrates the major steps of the proof.

Step 1 - acyclic between two operations: to prove a conflict graph with any two operations in this framework is acyclic. Based on the lock durations illustrated in Fig. 5, obviously a conflict graph with any two same operations is acyclic. Considering two different operations, a query report or a garbage clean cannot cause a cycle in a conflict graph with another operation, because the locks requested by a query report or a garbage clean are maintained until its commit point. Based on Fig. 5, an *OLU* and a *QLU* can cause potential cycle in a conflict graph, because these two operations may involve the same object and query. However, because of the conditional locks applied in the algorithms, if an *OLU* realizes that the query affected by the object is locked by a *QLU*, it will not access that query or update the corresponding query result. The same rule applies to the potential conflict on objects in a *QLU*. Because if two operations conflict on objects, they must conflict on queries too; no edges for object locations can be drawn between an *OLU* and a *QLU*. Therefore, in a conflict graph contains an *OLU* and a *QLU*, the only edge, if exists, can be drawn either from the *OLU* to the *QLU* or from the *QLU* to the *OLU* for query locations and query results. In other words, no cycle could occur in a conflict graph that consists of two operations.

Step 2 – acyclic among n operations: to prove given that a conflict graph with *n* operations ($OP_1, \ldots OP_n$) is acyclic, the conflict graph with operations ($OP_1, \ldots, OP_n, OP_{n+1}$) is also acyclic. Based on the proof in Step 1, if OP_{n+1} is a query report or garbage clean, it will not cause any new edges in the graph.

Suppose OP_{n+1} is an *OLU*, a possible edge from OP_i ($1<=i<=n$) to OP_{n+1} can be drawn for query results if OP_i is another *OLU*, or drawn for query locations and query results if OP_i is a *QLU*. Similarly, a possible edge from OP_{n+1} to OP_j ($1<=j<=n, j!=i$) can be drawn for query results if OP_j is another *OLU*, or drawn for query locations and query results if OP_j is a *QLU*. Now we prove there is no path from OP_j to OP_i by contradiction. Assume there exists any path P_{ji} from OP_j to OP_i, because the locks on one lockable structure are granted at the same time, P_{ji} cannot contain any edge drawn for query locations and query results. However, based on the analysis in Step 1, the edges between any two operations can only be query locations and query results. This contradiction shows that the existence of P_{ji} is impossible. Therefore, in case OP_{n+1} is an *OLU*, the conflict graph with operations ($OP_1, \ldots, OP_n, OP_{n+1}$) is acyclic.

Similarly, if OP_{n+1} is a *QLU*, a possible edge from OP_i to OP_{n+1} and a possible edge from OP_{n+1} to OP_j can be drawn for query locations and query results. Assume there is a path P_{ji} from OP_j to OP_i, P_{ji} cannot contain any edge drawn for query locations and query results. From the analysis in Step 1, if there is a path between OP_j and OP_i, all the edges on this path have to be drawn for query locations and query results. Because this contradiction shows that the existence of P_{ji} is impossible, the conflict graph with operations ($OP_1, \ldots, OP_n, OP_{n+1}$) is acyclic if OP_{n+1} is a *QLU*. Therefore, given that a conflict graph with *n* operations is acyclic, the conflict graph with *n+1* operations is acyclic, too.

Based on the above two steps, we can conclude that the concurrent operations supported in the proposed approach are serializable.

Q.E.D.

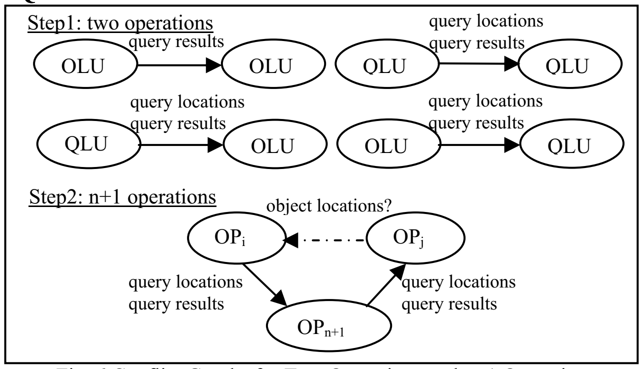


Fig. 6 Conflict Graphs for Two Operations and n+1 Operations.

Consistency: For either *O-tree* or *Q-tree*, the DGL approach (*ON* and *QN* in Object Location Update and Query Location Update) has been proved to protect the consistency on the R-tree. Furthermore, from the above serializable isolation analysis, each proposed operation keeps locking its target items (object/query) throughout the process, which ensures that the intermediate status between any two phases will not be accessed by other operations. Because the query report locks the query (*QR* in Query Report) and objects (*OD* in Query Report) from initiation to termination, only the results of all the operations committed before its initiation will be accessed. This guarantees the continuous query results are consistent with the current database.

Deadlock-freedom: Deadlock-freedom is assured as long as common sources are not accessed in opposite orders. Each indexing tree is deadlock-free internally with the protection of granular locking (*ON* and *QN* in Object Location Update and Query Location Update). The operations among multiple indices are proven to be deadlock-free in the following lemma.

Lemma 2: Object location updates (*OLU*), query location updates (*QLU*), query reports, and garbage cleans are deadlock-free given that any sub-operations involve a single indexing tree are deadlock-free.

Proof: Because query reports and garbage cleans only request locks at the beginning and release them at the commit point, these operations do not cause any deadlock with any other operations. We discuss *OLU* and *QLU* by observing the lock durations in Fig. 5 from the aspects of accessing objects, queries, and objects and queries.

Objects – Because in *OLU*, *ON* is placed together with *OD*, and in *QLU*, *ON* is placed before *OD* and released during *OD*, locks on the *O-tree* nodes and the *D-buffer* of the *O-tree* are not granted in opposite orders. Therefore, locks on objects are deadlock-free.

Queries – Similarly to locks on objects, *QN* is granted with *QD* in *QLU*, and *QN* is placed before *QD* and released during *QD* in *OLU*. In addition, *QD* always occurs before *QR* and is released during QR. Therefore, locks on the *Q-tree* nodes, the *D-buffer* of the *Q-tree*, and the *Q-results* are not requested in opposite orders. In other words, locks on the queries are deadlock-free.

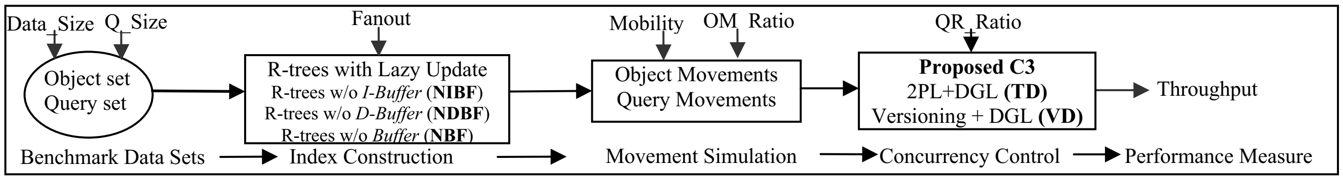


Fig. 7 Experiment Design.

Objects & queries - Note that *OLU* accesses objects before queries, while *QLU* accesses queries before objects. Therefore, the *O-tree* and the *Q-tree* are accessed in these operations in two opposite orders. However, based on the algorithms, conditional locks are requested on the second indexing tree accessed in both *OLU* and *QLU*. Once a conflict occurs on the second tree access, this tree access will be cancelled to eliminate the conflict. Therefore, the possible deadlocks caused by accessing two trees in opposite orders are prevented by the conditional locks that cede the conflicted objects or queries.

Based on the above analysis, the proposed concurrency control protocol is deadlock-free, given that any sub-operation on a single index is deadlock-free.

Q.E.D.

Summarizing the above, this concurrent access framework provides serializable isolation, consistency and deadlock-freedom. Therefore, it works correctly from the view of concurrency control.

VI. EXPERIMENTS

To evaluate the performance of the proposed framework, experiments on benchmark datasets have been conducted by measuring the throughput (number of concurrent operations processed in a second). The experiment design is illustrated in Fig. 7. The benchmark datasets were simulated by a network-based moving objects generator [23] using the road network of the City of Oldenburg. We set three classes of moving objects and queries to represent vehicles, bicycles, and pedestrians. Half of the initial moving objects generated by the generator were used as moving objects, and the second half of the objects were expanded to range queries. Based on the moving object set and moving query set, two 3-level R-trees were constructed with a fanout of 100. Meanwhile, the object movements simulated by the generator were translated into object location updates and query updates. These location updates and a set of random query report operations were submitted to the system as a multi-thread batch job. The overall execution time for each batch job was collected, and the system throughput was recorded by averaging the throughput of twenty batches of concurrent operations.

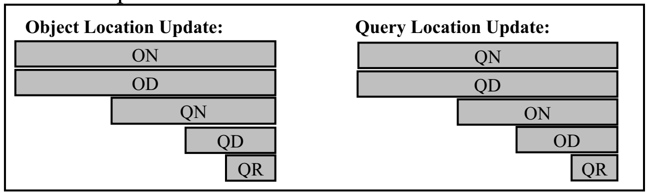


Fig. 8 Lock Durations for TD.

For performance comparison, one approach that fuses the record-oriented 2-phase locking transaction management approach with DGL on the R-tree [19] for concurrent operations, namely TD, has been implemented. Another approach integrates a record-oriented versioning approach with DGL, namely VD, has also been developed. TD and VD follow the continuous query processing approach in C3, except that the operations in TD/VD are executed using the 2-phase locking/versioning strategy among indices and DGL within the R-trees. The lock durations of object location update and query location update in TD are illustrated in Fig. 8. It inherits the complete indexing framework from C3, including *O-tree*, *Q-tree*, *D-buffer*, *I-buffer*, and *Q-result*. Therefore, its number of I/O accesses is as optimal as C3. Similarly, VD follows the same query processing algorithms as C3, except it requires redo operations when conflictions are detected. The conditional locks requested in location updates result in less redo operations in VD than pure versioning protocols, because they allow the operations continue to commit. In other words, TD and VD are both advanced approach for concurrent continuous query processing, which can achieve the same performance as C3 in single-user environments. Because there is no existing concurrent continuous query processing approach in literature, TD and VD are appropriate baselines with serializability for comparison. In addition, simplified versions of C3 without operation buffers have been developed to evaluate the impact of the *I-buffer* and *D-buffer*. Specifically, three versions, including C3 without *I-buffer* (NIBF), C3 without *D-buffer* (NDBF), and C3 without any buffer (NBF), were adopted for comparison.

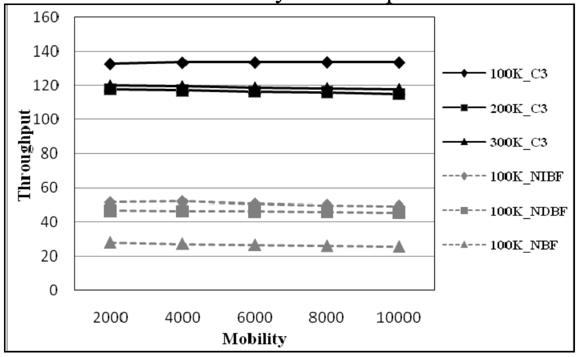
In the experiments, five parameters varied to simulate different application scenarios and to demonstrate their respective impacts. These parameters are listed as follows.

- **Data_size**: the number of initial moving objects and moving queries. It represents the number of moving objects plus the number of continuous queries.
- **Q_size**: the side length of query window for each moving query. It simulates query ranges in different applications. The default value was 5.
- **Mobility**: the total number of concurrent location updates for objects and queries in a batch. It corresponds to the frequency of object/query location updates. The default value was 2K.
- **OM_ratio**: the percentage of object location updates in Mobility. It reflects the relative update frequency between objects and queries. The default value was 50%.
- **QR_ratio**: the portion of query reports compared to Mobility. It shows the frequency of query report operations. The default value was 5%.

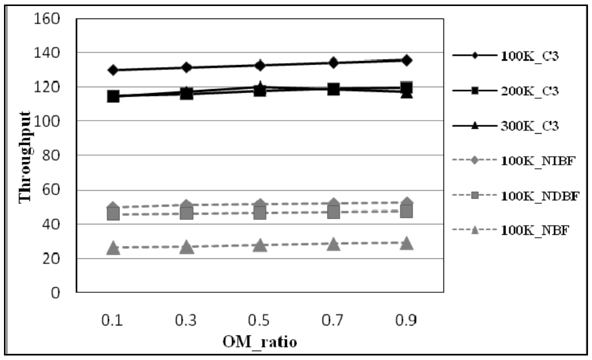
The proposed framework was implemented using JDK 1.5, based on the R-tree code from [24]. The experiment system was built on a Windows Server 2003 with two Duo-Core 2.4 GHz CPUs and 2 GB memory. Three sets of initial moving objects and moving queries were used in all the experiments, with the data_size 300K, 200K, and 100K, respectively. The performance gain of C3 is largely determined by the number of CPU cores. The more CPU cores are available, the more simultaneous operations can be processed, and consequently, the more opportunity for increased performance by C3.

A. Throughput vs. Buffers

Experiments were conducted to evaluate the effectiveness of the *I-buffer* and *D-buffer* in the framework. Location updates and continuous queries were processed on the original C3, NIBF, NDBF, and NBF, respectively. The size of each *I-buffer* is 20 items (about 1KB), and the size of each *D-buffer* is 400 items (about 20KB). For the 300K dataset, there were less than 100 internal tree nodes. Therefore, the size of buffers in the experiments was less than 150KB. Similarly, given a 3MB buffer, this setting can support more than 10 million moving objects. Since the mobility and OM_ratio are expected to have significant impacts on the system throughput, these two parameters were varied to analyze the impacts of the buffers.



a) Impacts of Buffers over Mobility



b) Impacts of Buffers over OM_ratio

Fig. 9 Throughput vs. Buffers.

Fig. 9 a) shows the throughput of C3 on the three datasets and that of the simplified C3 on the 100K dataset when the mobility increased from 2K to 10K. The *x*-axis shows the mobility, and the *y*-axis indicates the throughput. Generally, deactivating any operation buffer significantly increased I/O operations for tree updates. On the other hand, the increased updates on the R-trees caused additional locks and lengthened the lock durations. As shown in both figures, by deactivating *I-buffer*s, the system throughput decreased by more than 62% for the 100K dataset. When *D-buffer*s were deactivated, the system lost about 65% of the performance. When there was no operation buffer applied, the system throughput degraded more than 79% from the original C3 for the 100K dataset. As observed from the above results, *D-buffer*s promoted the system performance slightly more than *I-buffer*s. This is because the insertions with *I-buffer* need to traverse the R-tree, although only the higher levels for most of the time, and the *I-buffer*s close to the R-tree root may become bottlenecks. On the other hand, the deletions with *D-buffer* do not require tree traversal in most cases.

The comparison among different versions of C3 when the OM_ratio gradually increased is illustrated in Fig. 9 b), where the *x*-axis represents the OM_ratio and the *y*-axis shows the throughput. Following the trend of the original C3 on the 100K dataset, these simplified versions of C3 increased along with the OM_ratio. In addition, the NIBF outperformed the NDBF, and the NBF always had the lowest throughput.

Compared to the R-trees with buffers for frequent updates [7], [8], the C3 handles concurrent continuous query processing with seemingly lower throughput. It is because that each transaction in C3 consists of a location update, a result update, and a costly search, whereas a transaction in the related work contains only one operation which is usually an inexpensive update (costs less than 15% I/O of a search [8]). Considering this fact, C3 achieved comparable performance to the popular location update approaches. Furthermore, serializable isolation usually significantly degrades the system performance with its restricted locking policy, and C3 handled this efficiently.

B. Throughput vs. Mobility

In this set of experiments, the mobility of the objects and queries varied from 2K to 10K updates per batch, while the OM_ratio, QR_ratio, and Q_size were set to their default values. The throughput of the framework was measured on three datasets with different sizes, from 150K objects with 150K queries to 50K objects with 50K queries. The throughput of TD and VD on the same datasets and movements was also collected. The experiment results are shown in Fig. 10, where the *x*-axis represents the mobility and the *y*-axis shows the throughput. For clear comparison, the charts in the rest of this paper set the minimal *x*-value to 60. The throughput on all datasets decreased linearly when the mobility increased. These results are considered as reasonable because a higher mobility indicates more location update operations in the processing queue, and a longer queue leads to longer waiting time for each operation.

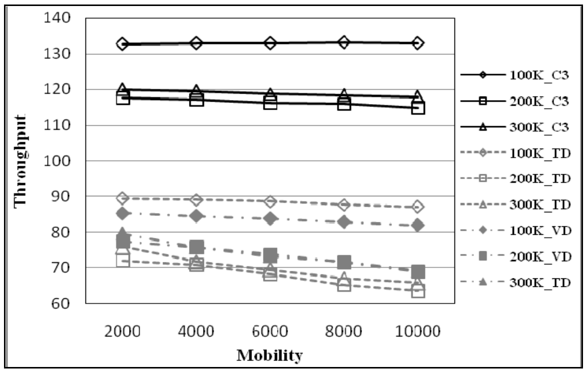


Fig. 10 Throughput vs. Mobility.

The throughput of C3 on both the 200K and 300K datasets reduced about 3% when the mobility changed from 2K to 10K. However, the decreasing rate on the 100K dataset was negligible with the increasing of the mobility. This suggests that mobility affects less on the smaller datasets than the larger ones. It is because concurrent operations on a smaller dataset may be processed quickly enough before causing conflicts.

Compared to the throughput of C3, TD on the three datasets processed 43~52 less operations every second, and VD processed 40~51 less operations. Specifically, the throughput on the 200K and 300K datasets lost about 38~44% by applying TD, and lost 34~39% by applying VD. The throughput on the 100K dataset dropped about 33% for TD and 36~39% for VD. Among the three approaches, TD decreased its performance most significantly when the mobility increased, because a frequently updated dataset benefits more from reduced lock durations. Compared to VD, TD performed worse on the large datasets. On the other hand, the decreasing rates of VD's throughput were higher than those of C3. The low throughput of VD was caused by the large number of redo operations during frequent updates, and these redo operations may consequently cause additional conflictions.

Interestingly, C3 on the 300K dataset performed about 3% better than on the 200K dataset. These results demonstrated the scalability of the R-tree and the advantages of the lazy group update technique. The lazy group update approach minimizes the cost of R-tree update operations, and concurrent operations are facilitated by the finer lockable granules on the R-trees with larger datasets. These advantages compensated the increased storage and overlaps among the tree nodes of the 300K dataset.

C. *Throughput vs. OM_ratio*

In this set of experiments, three datasets with data_size 300K, 200K, and 100K, were used to evaluate how the OM_ratio affects the system throughput. Fig. 11 illustrates the throughput of C3, TD, and VD, with the OM_ratio varied from 10% to 90%. The *x*-axis indicates the OM_ratio, and the *y*-axis represents the throughput. In most cases, when the portion of the object location updates in simultaneous operations increased, the throughput increased too. This is because an object location update usually costs less than a query location update. An object location update, based on the algorithm, performs a point insertion and a point query. On the other hand, a query location update inserts a window and processes a window query on the R-tree. Therefore, a query location update requires more I/O accesses and index locks.

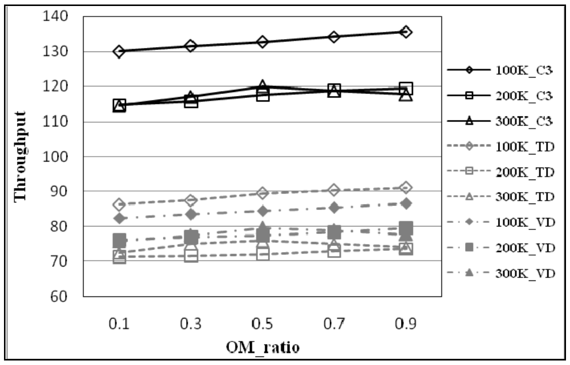


Fig. 11 Throughput vs. OM_ratio.

Furthermore, from this figure, it is clear that the 200K and 100K datasets benefited more from increasing the OM_ratio. The throughput of the 200K and 100K datasets raised 5% by increasing the OM_ratio, while that of the 300K dataset slightly decreased when the OM_ratio gradually increased from 50% to 90%. This difference was caused by the fact that in a larger dataset, the increased number of object location updates resulted in more conflicts with *X-lock*s on the *O-tree*, which compensated the benefit from fewer range query and update operations. These results show that the performance of location management on a small dataset can be significantly improved by increasing the OM_ratio. Similar to the previous set of experiments, the throughput of the 300K dataset was slightly better than the 200K dataset most of the time.

The throughput of TD and VD approaches in the figure shows significant performance degrade from C3. On all the three datasets, the throughput of TD was about 45 operations per second lower than C3, and VD lost 40~48 per second from C3. On all the datasets, the performance of TD and VD followed the trends of the corresponding C3 performance. The TD and VD on the 300K dataset decreased the throughput after the OM_ratio reached 50%, because write-write conflicts on the *O-tree* were increased, which caused longer waiting in TD and more re-do operations in VD. On the other hand, the performance of VD was lower than TD on the 100K datasets and better than TD on the 200K and 300K. This illustrated that VD had better scalability than TD, although performed worse on small datasets.

D. *Throughput vs. QR_ratio*

This set of experiments examines the relationship between the QR_ratio and the system throughput. The throughput was measured while the QR_ratio increased from 5% to 25%. The results are illustrated in Fig. 12, where the *x*-axis indicates the QR_ratio and the *y*-axis shows the throughput. Generally, a higher QR_ratio decreases the system performance, because more query reports are issued to consume the system resources. As shown in the figure, the throughput of C3 on the three datasets decreased by 0~2 operations per second when the QR_ratio increased from 5% to 25%. These results suggest that the cost for query report operation is relatively low, so that it can be efficiently processed without significantly blemishing the system performance. This is the benefit from the design of this concurrent continuous query processing, because the *Q-result* always stores the correct results, and the query report operation only requests *S-lock*s on the corresponding *Q-result* entry and the *D-buffer* entries of the *O-tree*.

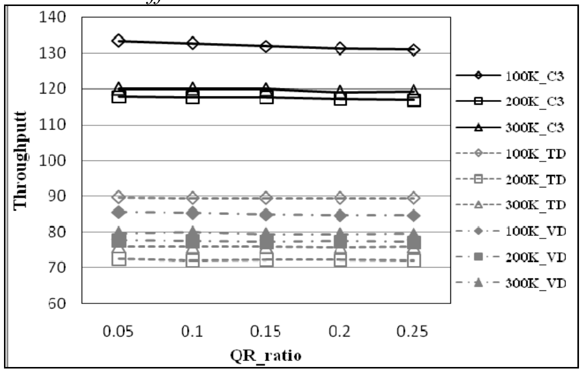


Fig. 12 Throughput vs. QR_ratio.

Similarly, although the number of query report operations in each batch increased from 100 to 500, there was no significant change on the throughput of TD and VD. As shown in the figure, C3 improved the performance by 50~55% from VD, and by 47~63% from TD.

E. Throughput vs. Q_size

This set of experiments varies the Q_size to study how the query window size affects the system performance. The experiment results are plotted in Fig. 13, where the *x*-axis shows the Q_size and the *y*-axis represents the throughput. The throughput of C3 on all the three datasets slightly and linearly decreased by about 7 when the Q_size increased from 5 to 25. Once the Q_size increases, each continuous query may cover more objects, and each object movement may affect more queries. Therefore, not only the tree access cost, but also the number of requested locks will increase. Following the trend in the previous experiment results, the 300K dataset performed better than the 200K dataset under C3 due to its fewer lock conflicts.

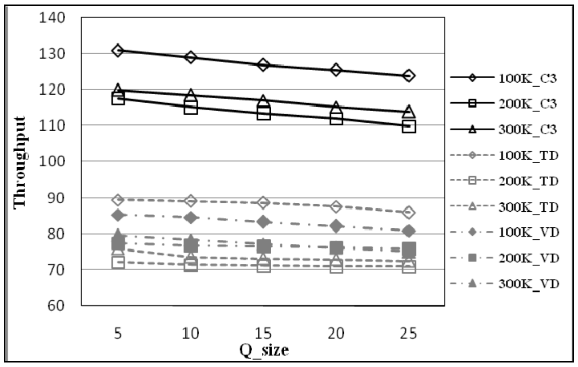


Fig. 13 Throughput vs. Q_size.

Similar to C3, the performance of TD slightly degraded when the Q_size increased from 5 to 25. C3 on each dataset achieved a significant performance improvement against TD. C3 on the 100K dataset improved the throughput by 47~44% from TD, 56~62% on the 200K dataset, and 58% on the 300K dataset. Compared to VD, C3 improved the performance by 53% on the 100K dataset, 45~52% on the 200K dataset, and about 50% on the 300K dataset.

The experiment results demonstrated that the performance of the proposed concurrent continuous query processing approach is efficient and scalable. As an interesting observation, in all these experiments, the 300K dataset outperformed the 200K dataset in C3, which demonstrated the scalability of the proposed approach in terms of data_size. Meanwhile, the OM_ratio, mobility and Q_size had noticeable impacts on the system throughput, whereas the QR_ratio did not significantly affect the performance. In addition, C3 gained substantial benefits by applying optimal lock durations and utilizing the operation buffers in the framework.

VII. Conclusion

This paper proposes C3, a concurrency control protocol for continuous queries, on an R-tree-based indexing structure. It is the first concurrency control protocol that protects the concurrent continuous query processing with lazy update techniques. It is proved to achieve serializable isolation, consistency, and deadlock-freedom for continuous queries over moving objects. Extensive experiment results on benchmark datasets have validated the efficiency and scalability of the proposed framework. This work provides an effective solution for continuous query processing and promotes its applicability in multi-user systems.

Concurrent operations involving a large portion of data, such as continuous kNN search and spatial join, still lack for efficient solutions. Future efforts could focus on extending C3 for these operations, and for indexing structures with velocity information for continuous query processing.

References

[1] (2008) Ride Finder. [Online]. Available: http://labs.google.com/ridefinder.

[2] (2008) Smart Trek. [Online]. Available: http://www.its.washington.edu/projects/strek.html.

[3] (2008) Flight Tracker - Real Time Airline Flight Tracking Software. [Online]. Available: http://www.airnavsystems.com/.

[4] (2008) ADS Aerospace Limited. [Online]. Available: http://www.adsaerospace.com/products/track.html.

[5] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, et al., "Supporting Frequent Updates in R-Trees: A Bottom-Up Approach," *in Proc. International Conference on Very Large Data Bases,* pp. 608-619, 2003.

[6] L. Biveinis, S. Saltenis, and C. S. Jensen, "Main-memory Operation Buffering for Efficient R-tree Update," *in Proc. International Conference on Very Large Data Bases*, pp. 591-602, 2006.

[7] B. Lin and J. Su, "Handling Frequent Updates of Moving Objects," *in Proc. ACM International Conference on Information and Knowledge Management,* pp. 493-500, 2005.

[8] X. Xiong and W. G. Aref, "R-trees with Update Memos," *in Proc. IEEE International Conference on Data Engineering*, pp. 22-31, 2006.

[9] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *in Proc. ACM SIGMOD International Conference on Management of Data*, pp. 321-330, 2004.

[10] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu, "Processing Moving Queries over Moving Objects Using Motion-Adaptive Indexes," *IEEE Transactions on Knowledge and Data Engineering,* vol. 18, No. 5, pp. 651-668, May 2006.

[11] H. Hu, J. Xu, and D. L. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects," *in Proc. ACM SIGMOD International Conference on Management of Data*, pp. 479-490, 2005.

[12] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, et al., "The Lowell Database Research Self-Assessment," *Commun. ACM,* vol. 48, No. 5, pp. 111-118, May 2005.

[13] J. Dai, C.-T. Lu, and L.-F. Lai, "A Concurrency Control Protocol for Continuously Monitoring Moving Objects," *in Proc. International Conference on Mobile Data Management*, pp. 132-141, 2009.

[14] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *in Proc. ACM SIGMOD International Conference on Management of Data*, pp. 47-57, 1984.

[15] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *in Proc. ACM SIGMOD International Conference on Management of Data*, pp. 322-331, 1990.

[16] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A Dynamic Index for Multi-dimensional Objects," *in Proc. International Conference on Very Large Data Bases*, pp. 507-518, 1987.

[17] J. K. Chen, Y. F. Huang, and Y. H. Chin, "A Study of Concurrent Operations on R-Trees," *Information Science,* vol. 98, No. 1, pp. 263-300, May 1997.

[18] V. Ng and T. Kamada, "Concurrent Accesses to R-Trees," *in Proc. Symposium on Large Spatial Databases*, pp. 142-161, 1993.

[19] K. Chakrabarti and S. Mehrotra, "Dynamic Granular Locking Approach to Phantom Protection in R-trees," *in Proc. IEEE International Conference on Data Engineering*, pp. 446-454, 1998.

[20] C.-T. Lu, J. Dai, Y. Jin, and J. Mathuria, "GLIP: A Concurrency Control Protocol for Clipping Indexing," *IEEE Transactions on Knowledge and Data Engineering,* vol. 21, No. 5, pp. 714-728, May 2009.

[21] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 5 ed.: Addison Wesley, 2007.

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*: Addison-Wesley, 1987.

[23] T. Brinkhoff, "A Framework for Generating Network- Based Moving Objects," *Geoinformatica,* vol. 6, No. 2, pp. 153-180, Jun. 2002.

[24] (2005) The R-tree Portal. [Online]. Available: http://www.rtreeportal.org.