2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware

A Concurrency Control Protocol for Continuously Monitoring Moving Objects

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

Lien-Fu Lai
Department of Computer Science and Information Engineering
National Changhua University of Education
Changhua City 500, Taiwan
lflai@cc.ncue.edu.tw

Abstract—**The increasing usage of location-aware devices, such as GPS and RFID, has made moving object management an important task. Especially, being demanded in real-world applications, continuous query processing on moving objects has attracted significant research efforts. However, little attention has been given to the design of concurrent continuous query processing for multi-user environments. In this paper, we propose a concurrency control protocol to efficiently process continuous queries over moving objects on a B-tree-based framework. The proposed protocol integrates link-based and lock-coupling strategies, and is proven to assure serializable isolation, data consistency, and deadlock-free for continuous query processing. Concurrent operations including continuous query, object movement, and query movement are protected under this protocol. Experimental results on benchmark data sets demonstrated the scalability and efficiency of the proposed concurrent framework.**

I. INTRODUCTION

Moving object management has attracted significant research efforts due to the wide usage of location-aware devices. The movement of vehicles, planes, and people can be traced, stored, and queried in moving object management systems. For example, vehicle tracking systems [1] track and display moving vehicles in real-time; flight monitoring systems [2] trace thousands of flying airplanes. Systems like these require spatial-temporal techniques to handle continuous moving objects. Among the tasks of moving object management, continuous queries keep refreshing the objects within the monitoring ranges of mobile queries. Examples of continuous query include "tracking all the patrol vehicles within 2 miles of the Inauguration Parade," and "reporting all the ships within 10 miles of this Coast Guard helicopter." Several continuous query processing techniques have recently been proposed [3-5].

The correctness of continuous queries on moving objects has to be assured by a well-designed concurrency control protocol. Figure 1 gives an example of inconsistent query results. In this example, a police vehicle Q keeps tracking all the buses within a given range of 0.5 mile. t_1 and t_2 are two consecutive query report timestamps. A and B are two buses 1 mile away from each other, driving in the same direction towards the police vehicle. 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 are submitted. Without a concurrency control protocol in place, these location updates and query reports in the database may exhibit inconsistent status. Some possible query result sets of Q at t_2 could be *Ø*, *{A}*, *{B}*, or *{A,B}*, within which only *{A}* is correct with respect to their actual locations at t_2. The situation that returns an empty result set at t_2 happens when Q is evaluated right after A deletes its old location, and the results of Q are reported before any other result updates related to Q. It is called pseudo disappearance [6], because the bus A seems disappeared in Q during its movement. B will be returned at time t_2 when B has updated its new location in the result set of Q, but Q's current results are reported before Q updates its new range. This scenario is called back order, where the query seems staying at its previous position while some objects have already updated their locations. 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. 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 will 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 in database, and both $\langle A,t_1 \rangle$ and $\langle A,t_2 \rangle$ intersect with $\langle Q,t_2 \rangle$. Further detailed discussion is provided in Section III. All the above inconsistent scenarios can be prevented by a well designed concurrency control protocol.

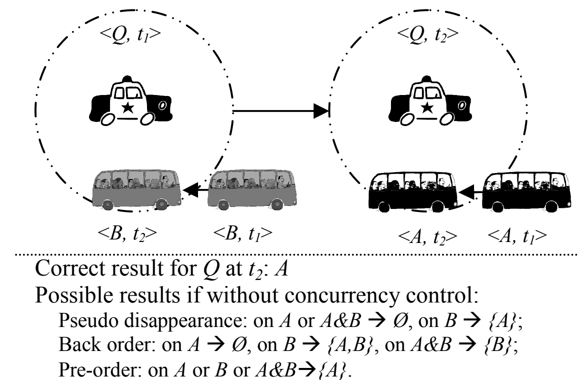


Figure 1. Inconsistency of continuous query w/o concurrency control.

Because of their efficient update and mature implementations, spatial access methods based on the B-tree are suitable for moving object management [7]. However, the continuous queries on moving objects bring two challenges to the concurrency control on B-tree-based databases: 1). *Continuous monitoring*: The data in query windows should be persistently secured to provide serializable isolation for concurrent operations; 2). *Frequent updates*: Locations of the objects in the database, as well as the query ranges, need to be updated frequently, which could cause read-write conflicts on the index and data pages. In order to efficiently process

978-0-7695-3650-7/09 $25.00 © 2009 IEEE
DOI 10.1109/MDM.2009.24

IEEE computer society

location updates and continuous queries, multiple indices can be applied to index objects, queries, and results accordingly [3]. The existing B-tree-based concurrency control techniques are not capable of providing serializable isolation on operations involving multiple indices. In addition, the popular versioning approaches for transaction management [8] are not feasible for frequent updates in moving object management.

This paper proposes a concurrency control protocol for concurrent spatial operations on moving objects based on the B-tree and Space-Filling Curves (SFC) [9]. It supports concurrent continuous query processing involving multiple indices, and avoids pseudo disappearance, back order, and pre-order. The major contributions of this work are as follows:

- Propose a concurrent continuous query processing approach for a B-tree-based framework;
- Fuse lock-coupling and link-based approaches to protect concurrent object updates, query updates, and continuous queries;
- Prove that serializable isolation, data consistency, and deadlock-free are guaranteed in the proposed framework;
- Validate the scalability and efficiency of the proposed concurrency control protocol by a set of extensive experiments on benchmark data sets.

The rest of this paper is organized as follows. Section II surveys the existing work on moving object management and concurrency control protocols. The preliminary of the proposed concurrent continuous query processing is discussed in Section III. Section 0 proposes the concurrent continuous query processing algorithms. The correctness proof is presented in Section V, and the experiment results are illustrated in Section VI. Finally, this work is concluded in Section VII.

II. Related Work

This section reviews existing techniques for B-tree-based moving object access methods, continuous query processing techniques, and concurrency control protocols for B-trees.

Benefitting from inexpensive update compared to R-trees, several spatial-temporal indexing structures [7, 10-12] based on B-trees have been proposed to manage moving objects and process spatial-temporal queries. Among these approaches, the B^x-tree [11] uses timestamps to partition the B+-tree, and each partition indexes the locations of the objects within a certain period. Because each moving object is modeled as a linear function of location and velocity, the B^x-tree can not only handle the queries on current locations, but also answer the spatial queries for the near future. The BB^x-tree [7] extends the indexing ability of the B^x-tree by supporting spatial queries on past locations. It applies a forest of trees; each tree corresponds to a certain time period. Queries with time and space constraints can be answered by the BB^x-tree. The B^{dual}-tree [12] improves the query performance of the B^x-tree on moving objects by indexing both the locations and velocities. Dual space transformation is applied in the B^{dual}-tree for efficient query access.

A straightforward approach to answer continuous queries is to process these queries as range queries periodically. However, this approach is not feasible when the number of continuous queries is large. Several approaches based on R-trees or hash tables have been proposed to answer the continuous moving range queries over moving objects by indexing both objects and queries. SINA [3] manages objects and queries by using hashing techniques, and incrementally processes positive and negative updates. Another approach, MAI [5], constructs motion-sensitive indices for objects and queries by modeling their movements, so that prediction queries for the near future can be supported. A generic framework for continuous queries on moving objects [4] has been proposed to optimize the communication and query reevaluation costs due to frequent location updates.

Concurrency control protocols [13-16] have been designed to ensure the consistency of B-trees under simultaneous non-spatial operations. These protocols can be categorized into link-based and lock-coupling approaches. The link-based protocols, represented by the B^{link}-tree [14, 15], apply only exclusive locks in update operations, and guarantee the consistency of concurrent operations based on the global order of data. In these approaches, each tree access follows the links from root to leaf and from left to right to identify a search path for the queried key. The lock-coupling approaches [13, 16] apply different lock types for read and write operations to provide flexible concurrency control. The lock-coupling approaches are able to provide serializable isolation to concurrent operations, albeit require comprehensive lock management on different levels of the index tree. In critical scenarios such as security systems and military applications, serializable isolation is required to ensure accurate relative positions among the query and objects. Although transaction management techniques can achieve serializable isolation, unlike concurrency control protocols for spatial indices, they tend to release the index nodes until commit point rather than as early as possible. Serializable isolation of the moving object management has recently been considered in CLAM [6], which provides concurrency control for location updates and queries on a B-tree-based spatial access framework. However, CLAM is not sufficient to support continuous queries, because it only works on a single indexing tree and for one-time queries.

The concurrent continuous query processing framework proposed in this paper applies the B^{link}-tree and hash techniques to construct indices for moving objects, moving queries, and query results, integrating multiple locking mechanisms for efficient concurrent access on objects and queries.

III. Preliminary

Before presenting the concurrent continuous query processing, we introduce the overall design of the proposed framework. In this framework, serializable isolation is supported on the concurrent spatial operations for continuous query processing. In other words, the results of these

concurrent operations are the same as the results of sequentially processing these operations ordered by their commit point. The access framework designed for scalable continuous query processing captures the current locations of the objects and queries.

To specifically describe the problem, several assumptions for the system environment are made as follows:

- **Point object**: Each moving object is represented as a spatial point; each object periodically reports its current location to the database.
- **Window query**: Each moving query is represented as a spatial box, which is the query window; each query periodically reports its new query window to the database.
- **Lock manager**: There exists a lock manager to support different lock types and maintain all the locks.

These assumptions are applicable in many real-world applications. Based on the above assumptions, a concurrent access framework for continuous queries is designed in the following sections. In this framework, the supported concurrent spatial operations are object movement, query movement, and query evaluation. **Object movement** takes object ID, old location, and new location as inputs, and updates the object index and the affected query results. **Query movement** takes query ID, old query window, and new query window as inputs, and updates the query index and results. **Query report** takes query ID as input, and outputs the set of objects covered by the query window. The output of a query report operation should reflect the current committed status. An overview of the proposed indexing structure and its concurrency control protocol is presented in the rest of this section.

A. Indexing Structure

To process the continuous query with efficiency and scalability, an indexing structure with one B^{link}-tree and two hash tables (*Q-table* and *R-table*) is applied to index current locations of both objects and queries. In this framework, the Hilbert Space-Filling curve [9], which preserves the spatial proximity of objects, divides the space into non-overlapped cells, and maps each object into a particular cell and each query window into a set of corresponding cells. Thus the spatial locations of objects/queries can be represented by one-dimensional cell IDs. The cell IDs of moving objects can then be indexed by a B^{link}-tree. Each entry in the leaf nodes of the B^{link}-tree points to the data page that stores the objects in its corresponding cell.

On the other hand, the cell IDs of moving queries are indexed using a hash table, *Q-table*, where the cell IDs are hash keys and the pointers to the corresponding queries are the contents stored in each bucket. The primary reason for using a hash table to index the queries, rather than another B^{link}-tree, is that point query is the only search on the query index in the proposed algorithms. In addition to these indices, another hash table, *R-table*, is used to store the query results in memory. In *R-table*, the query IDs are used as hash keys, and each entry stores a list of objects covered by a particular query.

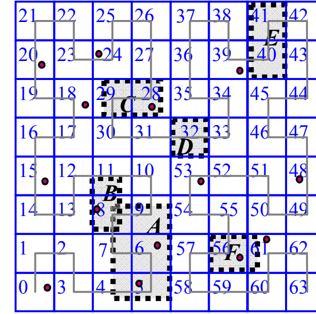


Figure 2. An example of objects and queries.

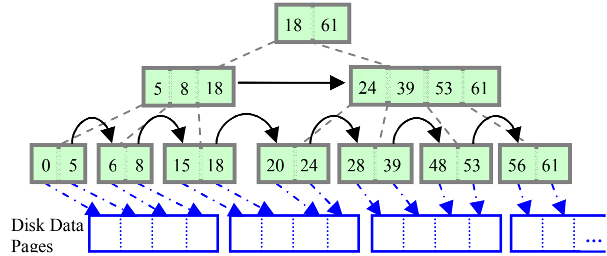


Figure 3. B^{link}-tree index for objects in Figure 2.

A snapshot example of moving objects and continuous queries is illustrated in Figure 2, where a 2D Hilbert curve with order of 3 is applied. There are six queries indicated by letters (*A, B, ... F*), and fourteen objects denoted by their cell IDs. Figure 3 demonstrates a B^{link}-tree constructed based on the objects in Figure 2. Each non-empty cell is indexed by a leaf node entry in the B^{link}-tree. The moving objects in one cell are stored in the same data page, or consecutive pages if overflowed. Different from the standard B-tree, in the B^{link}-tree, the tree nodes on the same level are linked from left to right for concurrency control purpose. The *Q-table* for query index corresponding to Figure 2 is shown in Figure 4. In the *Q-table*, each cell covered by any query has an entry. These entries consist of the corresponding queries, and can be randomly accessed by a given cell ID. Figure 5 shows the *R-table* for given objects and queries. Each entry of the *R-table* is associated with a query, and tracks the objects covered by that corresponding query based on the current database status. The objects in the example are denoted in the form of *O_cellID*.

Cell:	4	5	6	7	8	9	11	28	29	32	40	41	56	61
Query:	A	A	A	A	A B	A	B	C	C	D	E	E	F	F

Figure 4. Q-table for queries in Figure 2.

Query:	A	B	C	D	E	F
Object:	O_5; O-6	O_8	O-28			O_56

Figure 5. R-table for objects and queries in Figure 2.

In this indexing framework, an object movement will update its location in the B^{link}-tree, then search the *Q-table* for affected queries, and finally refresh the corresponding query results in the *R-table*. For example, if the object in the cell 53 is moving to the cell 32, which is covered by the query *D*, it first updates its location in the B^{link}-tree by

removing leaf entry 53 and adding new entry 32. Then the *Q-table* is searched and *D* is found to cover the object. At last, a record *O_32* is inserted into the *R-table* under the key *D*. A query movement needs to search the B^{link}-tree for the objects covered by its new query range, update the range in the *Q-table*, and refresh its query results in the *R-table*. For instance, query *E* is moving towards west by one cell to cover cells 38 and 39. The system searches the B^{link}-tree using cells 38 and 39, and identifies the object in cell 39. Then the *Q-table* is updated by removing entries 40 and 41, and creating two new entries 38 and 39 with *E* inside. The *R-table* is refreshed by inserting *O_39* to the entry *E*. A query report simply visits the entry in the *R-table* and outputs its object list.

B. *Concurrency Control Protocol*

Continuous query processing requires an appropriate concurrency control protocol to ensure correct results while objects and queries are moving. Taking the scenario in Figure 1 as an example, inconsistent results are caused by incorrect processing sequences. Suppose each movement can be decomposed into three atomic components: *D* for the deletion of an old location, *I* for the insertion of a new location, and *R* for refreshing the corresponding query results. In addition, let *qr* denote the query report for *Q* at t_2. A processing sequence contains … →*A.D*→*Q.R*→*A.I*→*qr* →*A.R*→... will output *null* as the results of *Q* at time t_2 (**pseudo disappearance** on *A*), because *A* disappears in database when *Q* updates its results, and there is no more update occurs before the query report at t_2. Another inconsistent result set *{B}* of *Q* at t_2 will be returned if the processing sequence contains … →*B.R*→*Q.D*→*qr*→*Q.I*→ … (**back order** on *A* and *B*). In this case, the bus *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. If a processing sequence contains …*B.R*→*Q.I*→*A.R*→*qr*→*Q.R* …, both *A* and *B* will be output as the results of *Q* at time t_2 (**back order** on *B*). That is because *B* adds itself into *Q*'s results based on *Q*'s old range, and *A* keeps itself in *Q*'s results based on *Q*'s new location. All these inconsistent processing sequences have to be prevented by the concurrency control protocol.

The concurrency control protocol for the proposed structure should support the concurrent spatial operations involving the B^{link}-tree, *Q-table*, and *R-table*. For efficiency and effectiveness, a concurrency control protocol combining link-based and lock-coupling strategies is adopted. The link-based strategy handles the operations on the B^{link}-tree, whereas the lock-coupling strategy ensures the consistency of the hash tables and between the index tree and hash tables. The lock-coupling strategy applies read-locks and write-locks on cells and queries, so that updates and searches can be isolated from each other. The proposed concurrency control protocol protects simultaneous operations from pseudo disappearance, back order, and pre-order without causing any deadlocks.

In order to provide serializable isolation on location updates and continuous queries, not only the objects, queries, and results, but also the empty SFC cells should be appropriately protected. Therefore, the locks on the B^{link}-tree nodes and continuous queries are not sufficient. Auxiliary structures need be employed to maintain the locks on cells.

To globally manage the locks in all the operations, as listed in TABLE 1, the lock manager maintains locks on three structures, tree nodes, queries, and cells. Locks on tree nodes, which assure the consistency of the tree via link-based strategy, are all write-locks for the B^{link}-tree update. Locks on queries (entries in *R-table*), which prevent inconsistency between queries and query results, contain read-locks and write-locks corresponding to each continuous query. Locks on the cells consist of object cell locks and query cell locks, which share the same SFC mapping but are independent to each other. The reason of employing these two sets of lock granules is to allow more concurrent accesses for better throughput. Read-locks and write-locks on cells for objects and queries are handled via two lock maps to prevent phantom access. The locks on the cells for objects are maintained by the Object Lock Map (OLM); the Query Lock Map (QLM) manages the locks on the cells for queries. Both lock maps are constructed with the same structure and size, as illustrated in Figure 6. In each lock map, each cell is associated with an integer to indicate the number of current read-locks granted for this cell. If a cell is being write-locked, -1 will be assigned to that cell in the lock map. In addition, a queue is also used by each cell to store pending cell locks, so that these processes can be notified once the cell is available.

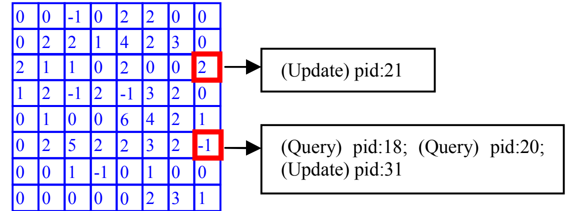


Figure 6. Example of lock map.

TABLE 1. TYPES OF LOCKS MAINTAINED.

	Tree Nodes	Queries	Cells	
			Objects	Queries
R-lock		√	√	√
W-lock	√	√	√	√
Protection	Inconsistent tree	Invalid results	Phantom access	Phantom access

In all these locks, read-locks are compatible with each other, while write-locks are exclusive to all the others. The tree nodes are protected by the link-based locking strategy, whereas the queries and cells are secured by the lock-coupling strategy. The spatial operations for continuous monitoring on moving objects integrated with these locks are introduced in detail in the next section.

IV. CONCURRENT CONTINUOUS QUERY

The proposed concurrent access framework supports object movement, query movement, and query report for scalable continuous query processing for moving objects.

A. Object Movement

The operation of object movement takes the object ID, old location, and new location of the moving object, as well as the B^{link}-tree, *Q-table* and *R-table*, as input parameters. This operation updates the object location on B^{link}-tree, identifies the queries that cover either the old location or new location from the *Q-table*, and refreshes the results of these queries in the *R-table*. Corresponding to the above subtasks, the object movement algorithm contains 3 phases, namely, object location update, query search, and result refresh.

The details of the algorithm are shown in Algorithm 1. **Phase 1**, **object location update**, first applies the SFC to find the cells that cover the old location or the new location of the object. After the cell IDs are determined, the algorithm traverses the B^{link}-tree to locate the leaf nodes that contain these cell IDs. Before modifying these leaf nodes, write-locks are requested for these leaf nodes to assure consistency on the tree and for the OLM cells involved in the movement to avoid phantom access. The tree update function determines whether the movement occurs within one cell, moves to an empty cell, or empties the original cell. Based on these situations, an optimized location update is then performed by releasing unnecessary locks on tree nodes as early as possible. The remaining write-locks on tree nodes are released by the end of this phase, while the locks on OLM cells are kept till the end of this process.

Phase 2, **query search**, retrieves the queries that cover the new location or old location of the object by looking up the *Q-table*. At the beginning of query search, read-locks are requested on the QLM cells involved in the movement. Thus the system can avoid phantom access on the query index. Then the *Q-table* is accessed to retrieve the candidate queries linked to these cell IDs. After refining the list of affected queries by computing their topological relation with the exact old location and new location, the queries that need to be updated are granted write-locks, and the read-locks on QLM cells can then be released. By the end of phase 2, the related continuous query results have been protected from being accessed by other simultaneous operations.

Phase 3, **result refresh**, modifies the entries in the *R-table* to refresh the query results. In this phase, the object ID is removed from the entries corresponding to the queries that the object is moving from, and is added into the entries for the queries that the object is moving to. After all the necessary updates are completed in the *R-table*, the write-locks requested in Phase 1 on the OLM cells and the write-locks requested in Phase 2 on the queries are all released. Once these locks are released, the involved object and queries become accessible to other operations.

Take the object movement example in Section III, and assume the object in the cell 53 is moving to the cell 32, covered by the query *D*. This algorithm first locates the leaf nodes that contain the cell 53 or will contain the cell 32, and requests write-locks on these nodes. The OLM then requests write-locks on cells 53 and 32. The entry for the cell 53 is deleted from the leaf node, and a new entry for the cell 32 is inserted, before the locks on these leaf nodes are released. In Phase 2, the QLM requests read-locks on cells 53 and 32 before the query *D* is retrieved. The algorithm then requests a query lock on *D* and releases the QLM locks. Finally, this object is inserted into the entry *D* in the *R-table*, and the query lock on *D* and the OLM locks on cells 53 and 32 are released.

```
Algorithm Object_Movement
Input: Oid: Object ID, loc_old: Old Location of Object, loc_new: New Location
of Object, T: Index Tree of Objects, Q: Q-table, R: R-table
Output: Nil

//Phase 1. Object location update
//locate on B^link-tree
c_old = SFC_map(loc_old); //determine the cell contains loc_old
c_new = SFC_map(loc_new); //determine the cell contains loc_new
n_old = T.traverse(c_old); // locate the leaf which contains c_old
n_new = T.traverse(c_new); // locate the leaf which contains c_new
T.writeLock(n_new ∪ n_old); // request tree node locks at one time
//modify B^link-tree
OLM.writeLock(c_old ∪ c_new); //request OLM cell locks at one time
T.update(c_new, n_new, c_old, n_old); //update B^link-tree if necessary for
    inserting loc_new and deleting loc_old
T.unWriteLock(n_new ∪ n_old); //release tree node locks
PageDeletet(n_old.entry(c_old), loc_old); //remove loc_old from the data page
    that contains cell c_old
PageInsert(n_new.entry(c_new), loc_new); //insert loc_new to the data page that
    contains cell c_new

//Phase 2. Query search
QLM.readLock(c_old ∪c_new); //request QLM cell locks at one time
q_old = Q.queries(c_old);
q_old = q_old.cover(loc_old); //identify queries cover loc_old
q_new = Q.queries(c_new);
q_new = q_new.cover(loc_new); //identify queries cover loc_new
R.writeLock(q_old∪q_new - q_old∩q_new); //request query locks at one time
QLM.unReadLock(c_old ∪c_new); //release QLM locks

//Phase 3. Result refresh
For each query q in q_old-q_new
        R.entry(q.Qid) -= Oid; //remove Oid from results
For each query q in q_new-q_old
        R.entry(q.Qid) += Oid; //insert Oid into results
R.unWriteLock(q_old∪q_new - q_old∩q_new);
OLM.unWriteLock(c_new ∪ c_old); //release OLM cell locks
Return:
```

Algorithm 1. Object movement.

B. Query Movement

The proposed operation of query movement updates the location of the given query in the *Q-table*, as well as the results of this query in the *R-table*, so that the database and query results are kept consistent. The query movement takes the query ID, old query window, new query window, and index structure as input parameters. This operation consists of three phases, object window search, query location update, and result refresh.

Algorithm 2 shows the details of the query movement operation. **Phase 1**, **object window query**, applies the SFC to locate the cells overlapped by the old query window and new query window. The B^{link}-tree is then traversed to retrieve all the objects covered by the new query window after read-locks are requested on the overlapped OLM cells. These read-locks are kept till the end of the process to avoid phantom access on these objects.

Phase 2, **query location update**, exclusively locks the QLM cells involved in the query movement, and consequently updates the corresponding entries of the *Q-*

table by adding or removing the query ID. This update assures concurrent object movements retrieve up-to-date query windows. After the query window is updated in the data page that stores this query, a write-lock is requested for the corresponding entry in the *R-table*, so that the results of this query are protected from being accessed. By the end of Phase 2, the write-locks on QLM cells are removed, because the query has been protected by the lock on the *R-table* entry.

```
Algorithm Query_Movement
Input: Qid: Query ID, win_old: Old Window of Query, win_new: New Window of Query, T: Index Tree of Objects, Q: Q-table, R: R-table
Output: Nil

//Phase 1. Object window searche
c_old = SFC_map(win_old); //determine the cells overlapping with win_old
c_new = SFC_map(win_new); //determine the cells overlapping with win_new
OLM.readLock(c_new); //request OLM cell locks at one time
o_new = T.rangeSearch(win_new); // find the objects overlapping with win_new

//Phase 2. Query location update
QLM.writeLock(c_old ∪ c_new); //request QLM cell locks at one time
For each cell c in c_old – c_old∩c_new
        Q.entry(c) -= Qid; //remove Qid from Q-table
For each cell c in c_new – c_old∩c_new
        Q.entry(c) += Qid; //remove Qid from Q-table
PageUpdate(Qid, win_new); //update query window
R.writeLock(Qid); //request query lock
QLM.unWriteLock(c_old ∪ c_new); //release QLM cell locks

//Phase 3. Result refresh
R.entry(Qid).objList = o_new; //update R-table entry
R.unReadLock(Qid); //release query lock
OLM.unWriteLock({c_new, c_old}); //release OLM cell locks
Return;
```

Algorithm 2. Query movement.

Phase 3, **result refresh**, replaces the results of the given query in the *R-table* with the objects retrieved in Phase 1. Thus the *R-table* can correctly reflect the current query locations and object locations. After the results of this continuous query are updated, the locks on this query (requested in Phase 2) and the affected OLM cells (requested in Phase 1) are released to allow access from other concurrent operations.

For instance, the following steps execute the concurrent query movement example in Section III, in which the query *E* moving towards west by one cell to cover cells 38 and 39. In Phase 1, the OLM places read-locks on cells 38 and 39 before the object in the cell 39 is retrieved via the B^{link}-tree. The QLM then requests write-locks on cells 38, 39, 40, and 41. The *Q-table* entries for cells 40 and 41 remove the query *E*, meanwhile the entries for cells 38 and 39 add *E*. By the end of Phase 2, query write-lock is placed on *E*, and the QLM locks are released. In Phase 3, the object in the cell 39 is inserted into the entry *E* in the *R-table*, before the query lock on *E* and the OLM locks on cells 38 and 39 are released.

C. Query Report

A query report takes a query ID and the *R-table* as input parameters, and returns the objects that are currently covered by this query. Due to the existence of the *R-table*, the query report process is simply to retrieve an entry from a hash table. Meanwhile, the query report will return the most recent results, because all the object and query movements refresh the affected query results in real time.

The concurrent query report operation is presented in Algorithm 3. At first, a read-lock is requested for the entry of the *R-table* based on the query ID. Then the content of this entry, a list of objects, is retrieved. At the commit point, the read-lock is released. In the proposed framework, the *R-table* is the only index component involved in this operation. The read-lock on the *R-table* has to cover the whole process to protect the corresponding entry from being updated by other concurrent operations.

Taking the objects and queries in Figure 2 as an example, if query report for the query *F* is issued, the algorithm first requests read-locks on the query *F*. The content of the entry *F* in the *R-table*, the object in the cell 56, is then retrieved. Finally, the lock on *F* is released before the commit point.

```
Algorithm Query_Report
Input: Qid: Query ID, R: R-table
Output: S: Set of Objects

R.readLock(Qid); //request query lock
S = R.entry[Qid]; //retrieve query results
R.unReadLock(Qid); //release query lock
Return S;
```

Algorithm 3. Query report.

V. CORRECTNESS

The proposed concurrent continuous query processing guarantees serializable isolation, consistency, and deadlock-free based on the well-designed protocol. **Serializable isolation** means the results of any set of concurrent operations are equal to that from the sequential execution of the same set of operations; **consistency** refers to the feature that the results always reflect the current commmited status; **deadlock-free** means any combination of the concurrent operations does not cause any deadlock. The correctness of the proposed protocol, in terms of these three features, can be validated by studying the lock durations for each operation.

Figure 7 shows the order and duration of the locks requested in each operation, including object movement, query movement, and query report. Each bar in the figure indicates a set of locks held on a particular sub-structure. The horizontal span of each bar in the figure represents the time period that the locks are granted on the corresponding structure. The overlaps of vertical projections of the bars indicate the intersections of the duration of the corresponding locks. The label of each bar shows the particular sub-structure to lock. Bars with label *BL* are the locks on the B^{link}-tree nodes, while *OL* and *QL* indicate the locks on OLM and QLM cells, respectively. Bars with label *RL* are the locks placed on the *R-table* entries to secure the visited queries. These locks on different sub-structures cooperate with each other to achieve serializable isolation, consistency, and deadlock-free.

Serializable isolation: The proposed object location update holds write-lock (*OL* in Object Movement) on the OLM cells to cover the old location and new location of the

object throughout the operation to occupy the cells related to this object exclusively. This operation also locks the affected queries (*QL* and *RL* in Object Movement) as soon as they are known till its commit point. Therefore, in the object movement, the object and corresponding queries are protected until the operation is completely processed. On the other hand, the query movement locks the overlapped OLM cells (*OL* in Query Movement) once they are known and till its commit point, and holds locks on the query window and query results (*QL* and *RL* in Query Movement) before updating the query and till the commit point. Thus the query, query windows, and affected objects are secured from conflicts. Finally, the query report operation holds locks on query results (*RL* in Query Report) all the time to avoid being accessed by other concurrent operations. From the above analysis, all these operations lock the target items before accessing them and till the end of process. This locking strategy guarantees that no conflicting access on common resources could occur among concurrent operations. Therefore, the proposed operations are serializable isolated.

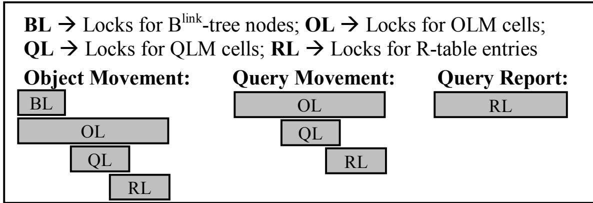


Figure 7. Lock durations for operations.

Consistency: The design of the B^{link}-tree, including the write-locks on tree nodes (*BL* in Object Movement) has been proved [16] to assure the consistency within the index tree. Besides the inner-tree protection, from the analysis for serializable isolation, each proposed operation locks its target items (object/query) throughout the process, which ensures the intermediate status will not be accessed by other operations. Because the query report locks the query results (*RL* in Query Report) from its initiation to termination, and the movement operations also locks the query results till their commit points, only the results of all the operations committed before the initiation of the query report will be retrieved. This guarantees the continuous query results are always correct with respect to the current database.

Deadlock-free: Deadlock-free is assured as long as the common sources are not accessed in an opposite order. The B^{link}-tree is deadlock-free internally [14], as long as the tree node locks (*BL* in Object Movement) are requested based on a global order. The locks in OLM, QLM, or *R-table* (*OL*, *QL* or *RL* in Figure 7) are deadlock-free, because the locks on any of these sub-structures in one operation are requested at the same time. The combination of these locks on different sub-structures will not cause any deadlock since they are requested in the same order. *OL* are placed before *QL*, and *QL* are before *RL* in this framework. Therefore, the proposed concurrency control protocol is deadlock-free.

VI. EXPERIMENTS

To evaluate the performance of the proposed framework, a set of extensive experiments on benchmark data sets have been conducted by measuring the throughputs (number of operations processed per second) of concurrent operations. The design of experiments is illustrated in Figure 9. The benchmark data sets were generated by a network-based moving objects generator [17] using the road network of City of Oldenburg, as shown in Figure 8. Three classes of moving objects and moving queries were set to represent vehicles, bicycles, and pedestrians. Half of the initial moving objects generated were used as moving objects, and the rest of the initial objects were expanded to range queries by specifying a certain size. To map the locations of the objects and queries to an one-dimensional space, the Hilbert curves with different orders were applied in the framework. Based on the moving object set and the Hilbert curve, a B^{link}-tree was constructed. On the other hand, 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 sent to the system as a multi-thread batch job. The overall processing time for each set of operations was collected to calculate the throughput.

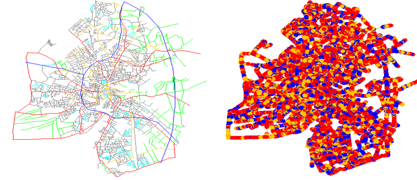


Figure 8. Road network of Oldenburg and data.

In the experiments, six parameters were varied to simulate different application scenarios and demonstrate their impacts on the system performance. These parameters are listed as follows.

- **Order**: the order of the Hilbert curve applied. It determines the number of cells for the whole space.
- **Data_size**: the number of initial moving objects/moving queries.
- **Mobility**: the total number of concurrent location updates for objects and queries issued in a batch.
- **OM_ratio**: the percentage of object location updates in Mobility.
- **QR_ratio**: the portion of query reports compared to Mobility.
- **Q_size**: the side length of query window for each moving query. It simulates query ranges in different applications.

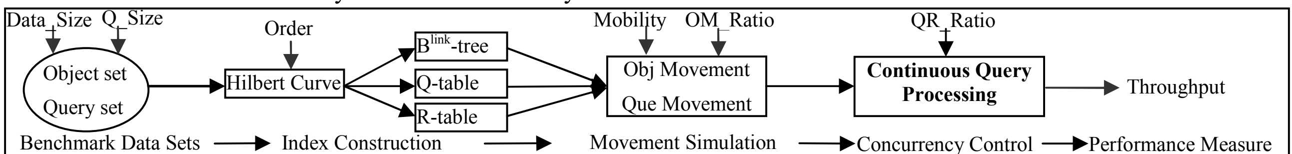


Figure 9. Experiment design.

The performance of the framework is evaluated by varying these parameters. The default settings and ranges of these parameters are listed in TABLE 2.

TABLE 2. EXPERIMENT PARAMETERS.

	Order	Data_Size	Mobility	OM_ratio	QR_ratio	Q_size
Default	8	N/A	2K	50%	5%	5
Range	8~12	50K~150K	2K~10K	10%~90%	5%~25%	5~25

The proposed framework was implemented in Java using JDK 1.5. The experiment system was built on a desktop with a Pentium-D 2.8 GHz CPU. Three sets of initial moving objects and moving queries were used in each set of experiments, with data_size 150K, 100K, and 50K, respectively. The average throughputs under different parameter settings were calculated by collecting the average processing time of ten batch executions. For comparison, the continuous query processing extended from CLAM [6] with serializable isolation (indicated as CI) was implemented. CI treats the suboperations on each single index as an item in a transaction. The CI operations under the proposed framework acquire locks before accessing all the required resources, except B^{link}-tree nodes, and release them at commit point. CI inherits the query processing of the proposed approach by fusing the link-based and lock-coupling locking strategies. The only difference is that CI does not optimize the lock duration for query locks and QLM locks. This CI approach applies the link-based locking on the B^{link}-tree, which has been shown to have fewer read/write conflicts with less maintenance overhead than lock-coupling protocols, therefore it can achieve higher throughputs than a pure lock-coupling approach. Note that the proposed approach was compared to a non-trivial method to demonstrate its advantages. Since there is no existing protocols to provide serializable isolation for continuous query processing, to the best of our knowledge, CI is the most matchable solution to compare with. The detailed experiment results are presented in the following subsections.

A. *Throughput vs. Mobility*

In this set of experiments, the impact of mobility was studied by capturing the throughputs of continuous query processing with different numbers of movements. Basically, a higher mobility means more objects and queries that report their movements at the same time. Consequently, as more movements need to be processed, the processing queue becomes longer and the queuing time for each movement will be increased. This can be verified by the results illustrated in Figure 10, where the *X*-axis indicates the mobility value and the *Y*-axis represents the system throughput. When the mobility increased from 2,000 to 10,000, the system throughputs on all the data sets decreased by more than 60%.

Comparing the results from different data sets, the smaller data set always performed better than the larger ones. The throughput of the 50K data set was about three times better than that of the 100K data set. Similarly, the performance of the 150K data set was about 15% worse than the 100K data set. The reason of these significant gaps is that a smaller data set requires a smaller B^{link}-tree, less data pages for each B^{link}-tree leaf entry, and smaller hash tables. Therefore, less I/O is consumed for a movement in smaller data sets.

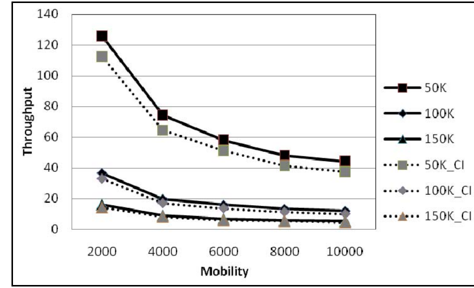


Figure 10. Throughput vs. mobility.

Another fact observed from this set of experiments is that the proposed approach performed 10~20% better than the CI method. This means the design of the proposed concurrency control protocol reached a higher concurrency level by optimizing the lock durations. This improvement can be further enhanced by running on more processing units.

B. *Throughput vs. OM_ratio*

This set of experiments demonstrated the trend of throughput when increasing OM_ratio. A higher OM_ratio means more object movements within a given mobility. The results are shown in Figure 11, where the *X*-axis represents OM_ratio and the *Y*-axis indicates the throughput. As observed from the figure, the performance of concurrent continuous query processing dropped significantly when OM_ratio increased from 10% to 90%.

When OM_ratio was set to 0.1, the system performed surprisingly well on all the three data sets. These high throughputs then decreased quadratically. This is because the object movement requires updating the B^{link}-tree, while the query movement only updates hash tables. Updates on a B^{link}-tree are costly compared to updating a hash table, because they not only require more I/O operations to locate the data page, but also have to perform node split/merge sometimes. Furthermore, one update operation on the *Q-table* only locks that single query, but the B^{link}-tree needs to lock nodes during updating, which involves multiple objects and causes more conflicts. These results justify the design of applying a hash table to index queries, and suggest that query movement in the proposed framework is more efficient than object movement.

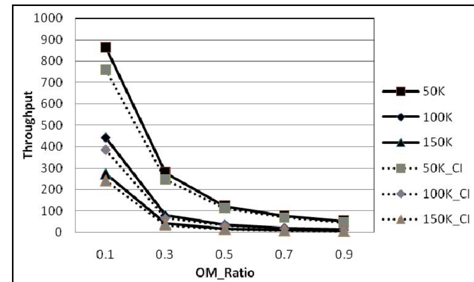


Figure 11. Throughput vs. OM_ratio.

The corresponding throughputs of CI approach showed a similar trend, but always lower than the proposed framework by 10~20%. The improvement was more significant when OM_ratio was low, because the proposed concurrency control protocol reduces the lock duration mainly for QLM and the locks on queries. Therefore, the performance of continuous monitoring with more query movements was promoted as expected.

C. Throughput vs. Q_size

The relationship between throughput and Q_size was studied in this set of experiments. Q_size was increased gradually from 5 to 25 to probe its impact on the system performance. The results are illustrated in Figure 12, where the *X*-axis represents Q_size and the *Y*-axis shows the throughput. In all the three data sets, although the smaller data sets outperformed the larger ones, the throughputs kept constant when Q_size increased. From this figure, obviously Q_size did not show much impact on the performance. The reason is that a query window with size 25*25 is still small comparing to the data space. With this query window, a B^{link}-tree search can most likely find the results within one data page. In this case, a range search requires the same I/O cost as a point search.

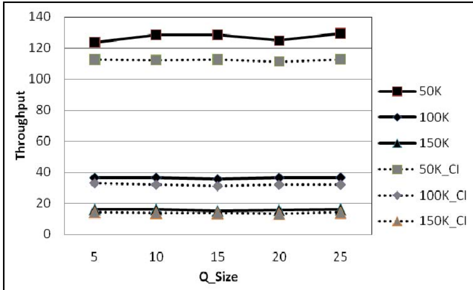


Figure 12. Throughput vs. Q_size.

Similarly, the throughputs of the data sets applying CI approach were constant when Q_size varied. The CI approach on each data set performed about 10~20% worse than the proposed concurrency control protocol on the same data set.

D. Throughput vs. QR_ratio

The focus of this set of experiments was to study the impact of QR_ratio on concurrent continuous query processing. QR_ratio was gradually increased from 5% to 25%, and the corresponding throughputs on three data sets were collected and compared. The results are plotted in Figure 13, where the *X*-axis indicates QR_ratio and throughput is represented in the *Y*-axis.

As shown in the figure, the throughputs of three data sets slightly decreased when QR_ratio raised significantly. When QR_ratio increased from 5% to 25%, the throughputs only reduced about 5%. These results suggest that the cost for a query report operation is negligible to the system comparing to object movement and query movement. These results can be well explained via the design of this concurrent continuous query framework. Query report, as illustrated in Algorithm 3, only requests a read-lock on the given query, and reads the corresponding entry in the *R-table*. It is a memory-based operation and can hardly block other operations.

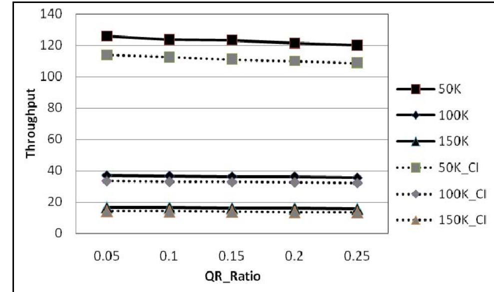
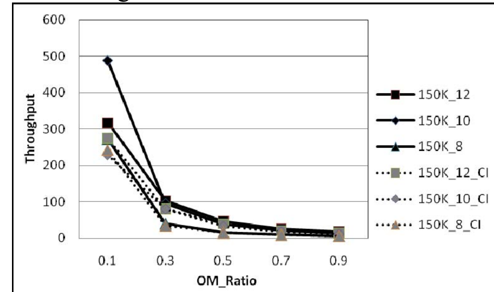


Figure 13. Throughput vs. QR_ratio.

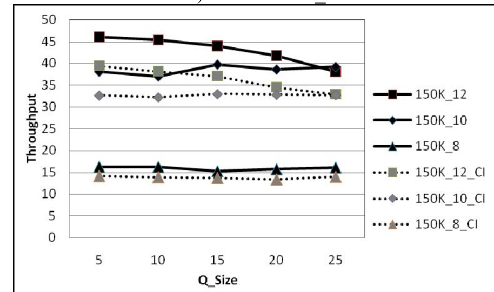
On the other hand, as expected, the performance of CI approach was always 10%~20% worse than the proposed approach, and decreased in the same speed as the proposed concurrent operations on the corresponding data set.

E. Throughput vs. SFC Order

This set of experiments demonstrate the impact of the order of SFC mapping on the performance of concurrent continuous query processing. The Hilbert curves with order 8, 10, and 12 were used to construct B^{link}-trees on the 150K moving object set. Obviously, a higher SFC order results in finer cells in space, and consequently more cell IDs will be indexed in B^{link}-tree. In other words, less objects will be contained in a single cell.



a) Over OM_ratio.



b) Over Q_size.

Figure 14. Throughput vs. SFC order.

OM_ratio and Q_size were varied in the experiments to investigate the impact of order in different scenarios. As illustrated in Figure 14 a), when OM_ratio increased from 0.1 to 0.9, the throughput of the system on different data sets

kept decreasing. In most of the cases, the Hilbert curve with order 12 performed better than order 10, and the curve with order 10 performed better than order 8. That is because: 1) When SFC order is higher, the B^{link}-tree has more nodes, and the locks on B^{link}-tree nodes have less chances to cause conflict; 2) High concurrency can be achieved by having more cells in OLM and QLM; 3) With a higher SFC order, there are less data pages associated with a B^{link}-tree leaf entry, leading to less I/O for moving object retrieval/update. Interestingly, when OM_ratio was 10%, the Hilbert curve with order 10 performed the best among the three orders. For the order 12 Hilbert curve with a small OM_ratio, the improvement from finer lock granules and more efficient B^{link}-tree data accesses was compensated by the additional cost of the Hilbert curve mapping function calculation.

Observing the performance of the corresponding CI approach, the CI on the Hilbert curve with order 10 did not reflect the advantage of finer lock granules as significant as the proposed approach, because of its increased lock durations caused by the Hilbert mapping calculation.

When Q_size was increased from 5 to 25 in Figure 14 b), the Hilbert curve with order 12 performed better than order 10 in most cases, and the curve with order 10 outperfomed order 8 all the time. Furthermore, the improvement from order 8 to order 10 was greater than that from order 10 to order 12. On the other hand, the Hilbert curves with order 8 and 10 performed constantly, behaving similarly as discussed in Subsection C. However, the Hilbert curve with order 12 exhibited a 20% performance drop in the figure when Q_size increased. These results suggest that the cell size in the Hilbert curve with order 12 is comparably small to the query sizes. Therefore, a query may need to scan multiple cells to locate the moving objects.

Concluded from the above experiment results, the proposed concurrent continuous query processing optimizes the locking strategy and improves the concurrency level. The parameters, including order, mobility, data_size, and OM_ratio are found having significant impacts on the performance of the proposed framework. Within these parameters, the increasing of mobility, data_size, or OM_ratio degraded the system performance, whereas a higher order promoted the throughput.

VII. CONCLUSION

This paper proposes a framework for concurrent continuous query processing based on the B-tree and SFC. Indices for moving objects, moving queries, and query results have been integrated to efficiently handle movements and query reports. The proposed concurrency control protocol optimizes the locking strategy and provides serializable isolation, data consistency, and deadlock-free. Its correctness has been proved by analyzing the lock durations of the operations, and the performance has been evaluated by a set of extensive experiments. This work provides the applicability of efficient continuous query processing in multi-user systems, and offers expandability to other B-tree-based moving object management approaches. Future efforts could be devoted to applying this framework to motion-based spatial-temporal databases, such as the B^x-tree and the BB^x-tree. Expanding this concurrency control protocol to R-tree-based indexing structures would also be an interesting direction.

REFERENCES

[1] Smart Trek, http://www.its.washington.edu/projects/strek.html, Accessed in May, 2008

[2] Flight Tracker - Real Time Airline Flight Tracking Software, http://www.airnavsystems.com/, Accessed in Jun., 2008

[3] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *in Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, Paris, France, 2004, pp. 321-330.

[4] H. Hu, J. Xu, and D. L. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects," *in Proceedings of the 24th ACM SIGMOD International Conference on Management of Data*, Baltimore, MD, USA, 2005, pp. 479-490.

[5] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu, "Processing Moving Queries over Moving Objects Using Motion-Adaptive Indexes," *IEEE T Knowl Data En,* vol. 18, pp. 651-668, May 2006.

[6] J. Dai and C.-T. Lu, "CLAM: Concurrent Location Management for Moving Objects," *in Proceedings of the 15th ACM International Symposium on Advances in Geographic Information Systems*, Seattle, WA, USA, 2007, pp. 292-299.

[7] D. Lin, C. S. Jensen, B. C. Ooi, and S. Šaltenis, "Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects," *in Proceedings of the 6th International Conference on Mobile Data Management*, Ayia Napa, Cyprus, 2005, pp. 59-66.

[8] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu, "Transaction Time Support inside a Database Engine," *in Proceedings of the 22nd IEEE International Conference of Data Engineering*, Atlanta, GA, USA, 2006, pp. 35 -46.

[9] H. Sagan, *Space Filling Curves*. Berlin, Germany: Springer, 1994.

[10] C. S. Jensen, D. Tielsytye, and N. Tradilauskas, "Robust B+-Tree-Based Indexing of Moving Objects," *in Proceedings of the 7th International Conference on Mobile Data Management*, Nara, Japan, 2006, pp. 12-20.

[11] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and Update Efficient B+-Tree Based Indexing of Moving Objects," *in Proceedings of the 30th International Conference on Very Large Data Bases*, Toronto, Canada, 2004, pp. 768-779.

[12] M. L. Yiu, Y. Tao, and N. Mamoulis, "The B^{dual}-Tree: Indexing Moving Objects by Space Filling Curves in the Dual Space," *VLDB J,* vol. 17, pp. 379-400, May 2008.

[13] V. W. Setzer and A. Zisman, "New Concurrency Control Algorithms for Accessing and Compacting B-trees," *in Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago de Chile, Chile, 1994, pp. 238-248.

[14] W. d. Jonge and A. Schiff, "Concurrent Access to B-trees," *in Proceedings of the 1st PARBASE International Conference on Databases, Parallel Architectures and Their Applications*, Miami Beach, FL, USA, 1990, pp. 312-320.

[15] P. Lehman and S. Yao, "Efficient Locking for Concurrent Operations on B-trees," *ACM T. Database Syst.,* vol. 6, pp. 650-670, Dec. 1981.

[16] I. Jaluta, S. Sippu, and E. Soisalon-Soininen, "Concurrency Control and Recovery for Balanced B-link Trees," *VLDB J,* vol. 14, pp. 257-277, Apr. 2005.

[17] T. Brinkhoff, "A Framework for Generating Network- Based Moving Objects," *Geoinformatica,* vol. 6, pp. 153-180, Jun. 2002.