DIME: Disposable Index for Moving Objects

Jing Dai
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY, USA 10532
jddai@us.ibm.com

Chang-Tien Lu
Virginia Polytechnic Institute and State University
7054 Haycock Road, Falls Church, VA, USA 22043
ctlu@vt.edu

Abstract—**Increasing usage of location-aware devices, such as GPS and RFID, has made moving object management an important task. Existing spatial-temporal indexing techniques support efficient queries on large number of moving objects. In these techniques, significant I/O is consumed by removing obsolete locations, which impairs the performance of moving object management. On the other hand, some techniques have been designed to index moving objects in main memory to facilitate frequent location updates. However, they are limited by the size of available memory. In this paper, we propose a generic spatial-temporal index framework, Disposable Index for Moving objEcts (DIME), to efficiently handle location management over mobile agents with hybrid storage support. The proposed disposable index framework eliminates delete operations on the spatial indexing structure and processes insert operations in memory only. Most existing spatial indexing structures can be adopted in this generic framework. Both snapshot and continuous query processing has been designed for this framework. Experimental results on benchmark data sets demonstrated the scalability and efficiency of DIME.**

Keywords-Spatial index; moving objects; continuous query

I. Introduction

Moving object management is now a popular research task due to the mature applications of location-aware devices. Equipped with GPS or RFID, flights, vehicles, pedestrians, and mobile sensors are able to continuously record and report their locations. These locations can be acquired, stored, and queried in moving object management systems for monitoring and analysis. These systems, including flight monitoring, vehicle management, and parental tracking, require efficient indexing structures to handle both snapshot and continuous queries. Snapshot queries retrieve moving objects based on their locations at a given timestamp, e.g., "find all the cell phone users in this park," and "report the nearest gas station." Continuous queries keep refreshing the objects within the monitoring ranges of mobile queries [4-6]. Examples include "tracking all the patrol vehicles within 2 miles of the Inauguration Parade," and "monitoring ships within 10 miles of this Coast Guard helicopter."

Frequent location updates raise challenges to indexing methods and query processing approaches, because real-time response is a critical measure for moving object management systems. Existing spatial access methods for moving objects, including R*-tree-based approaches [1-4] and linear spatial access methods [12-15], handle location updates by inserting new locations and deleting old ones. In other words, each location update needs to update the index twice. Although some buffering techniques [5-7] can be applied to group the delete operations, these deletions are eventually executed on the index and consume substantial I/O resources. Memory-based indexing techniques [8, 9] have been proposed to support high update frequency. However, their applicability is limited due to the competition on available memory with OS and other applications. We will discuss this in more details in Section II.

Fig. 1 shows an example of object location update, where the moving objects are indexed using an index tree. In this example, the movement of the object o_8 causes its deletion from the leaf node D and insertion into the leaf node E. Deleting the old location of o_8 from D results in merging nodes C and D, because the number of objects in D is now lower than the minimum capacity of 2. Consequently, this node merging operation is propagated upwards to nodes J and I, and then to nodes M and N, due to the lower level merging. In this case, 7 (shaded nodes in Fig. 1) out of the 15 tree nodes need to be reconstructed. Significant effort is devoted to removing the obsolete information, thus degrading the performance. By caching the delete operations [5], the update I/O cost could be dramatically reduced. However, these cached delete operations still need to revise the index structure at a certain time. We observed that further improvement can be achieved if the delete operations do not modify the index at all. On the other hand, existing moving object indexing approaches are lack of a good balance between the efficiency of memory-based indices and the scalability of disk-based indices.

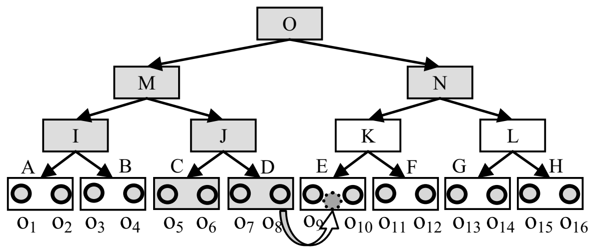
O
M
N
I
J
K
L
A
B
C
D
E
F
G
H
o_1 o_2 o_3 o_4 o_5 o_6 o_7 o_8 o_9 o_{10} o_{11} o_{12} o_{13} o_{14} o_{15} o_{16}

Fig. 1 An Example of Location Update

We propose an indexing framework, DIME (Disposable Index for Moving objEcts), to eliminate delete operations on the index structure and process insert operations only in memory. Since the locations of moving objects are frequently updated, location information can become obsolete quickly. For this scenario, we provide a solution to reduce the unnecessary I/O for delete operations. Instead of deleting the obsolete location for each location update, a whole chunk of the index will be detached without changing the internal structure. In addition, a hybrid storage model is applied in DIME so that the insert operations are only processed in the in-memory component of the index, and the majority of the index is still located on disk for search.

The basic idea is illustrated in Fig. 2. DIME consists of multiple components (e.g., indexing trees) allocated based on different time periods. Each component can be treated as an independent index for the objects updated in a respective time period. Only the most recent component is necessary to be stored in memory. There are three operations supported on each component: insert a location, search locations, and dispose a whole component. An object movement triggers an insertion on the index in memory, and flags its old location as obsolete. A search query traverses each in-memory and on-disk component to identify the result. Taking the object movement in Fig. 1 as an example, the new location of o_8 will be inserted into a current in-memory component ($T+3\Delta t$) of the index. Meanwhile, its old location will be kept in its original index node until that component ($T+\Delta t$) is disposed. Thus, DIME requires no modifications on the indexing trees for deletion and only processes insertion in the in-memory component.

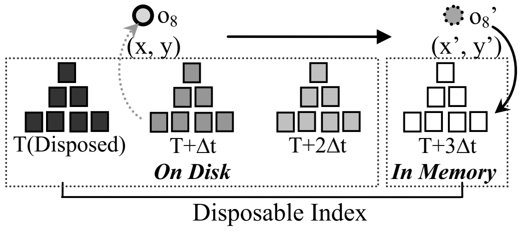

Fig. 2 Location Update on Disposable Index

The proposed moving object access framework is independent to the underlying indices applied on each component. Either spatial tree indices (e.g., the R-trees) or linear spatial indices (e.g., the B-trees with Space-filling Curves) can be adopted for DIME. This design allows applying appropriate indexing approaches for various application scenarios, and being adaptive to potential spatial index techniques in the future. In addition, this moving object index structure is natively sliced based on the update timestamps for hybrid storage management. The major contributions of this paper are as follows:

- **Proposal of a generic and efficient access framework for managing moving objects**: The disposable index is designed to reduce the I/O cost for location updates, so that frequent movements are efficiently supported;
- **Enabling hybrid storage for moving objects**: DIME consists of multiple indexing components, and allocates only the current component in memory, which handles expensive updates. The disk-based components contribute on search operations;
- **Extension of disposable index for continuous queries**: The proposed framework has been extended to process not only snapshot queries, but also continuous queries on moving objects;
- **Performance analyses and experiment evaluation**: Theoretical analyses and extensive experiments on benchmark datasets have been conducted to demonstrate the performance of the proposed index structure.

The rest of the paper is organized as follows. Section II reviews the existing work on moving object management. The preliminary of DIME is introduced in Section III. Section IV proposes the operation algorithms. The performance analysis is discussed in Section V, and the experiment results are presented in Section VI. Finally, this work is concluded in Section VII.

II. Related Work

This section reviews existing moving object access techniques, including indexing structure based on the R-trees and the B-trees, and the corresponding continuous query processing approaches.

As a popular multi-dimensional indexing structure, the R-tree family [2], including the R*-tree [1] and the R+-tree [4], provides a robust tradeoff between efficiency and implementation complexity. The R-trees are usually considered as costly for updating, which makes them unsuitable for processing moving objects. Many approaches utilizing hashing and lazy update techniques have been proposed to reduce the update cost of the R-tree and its variants. The Frequent Update R-tree (FUR-tree) [3] 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 from another aspect. The R-tree with update memos, RUM-tree [5], applies the main memory buffer to cache delete operations, so that they can be processed later when particular leaves are accessed. Lazy group update on the R-tree, LGUR-tree [6], 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 for both insert and delete operations [7].

Benefitting from inexpensive update compared to the R-trees, linear spatial indexing structures [10-13] based on B-trees and Space-Filling Curves (SFC) 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 objects within a certain period. Because each moving object is modeled as a linear function of location and velocity, the B^x-tree can handle the queries on current locations, as well as answer the spatial queries for the near future. The BB^x-tree [13] extends the indexing ability of the B^x-tree by supporting spatial queries for past locations. It applies a forest of trees; each tree corresponds to a certain time period. The B^{dual}-tree [12] improves the query performance of the B^x-tree by indexing both locations and velocities. Dual space transformation is applied in the B^{dual}-tree for efficient query access. The ST^2B-tree [14] provides the ability to partition space into SFC cells with different granularity, and the partition granularity is dynamically tuned based on the data distribution. The capacity of each SFC cell is balanced for better query performance.

Recently in-memory indexing approaches for moving object management [8, 9] have been proposed to efficiently handle location data streams. The availability of main memory is the main constraint to these approaches on large datasets, because a real data server needs to supply fundamental software and functions, including OS, management and maintenance components in DBMS, and indices of other data tables. For instance, the in-memory R-tree solution [8] requires a roughly 2GB R-tree for 100M objects (2% of mobile subscribers world wide), which may use up the available memory in a loaded 4GB Linux system. This issue becomes more critical when versioning techniques are applied for concurrency control. Using hash tables for updates in memory [9] cannot support spatial queries on the recently updated locations, which introduces potential inconsistency.

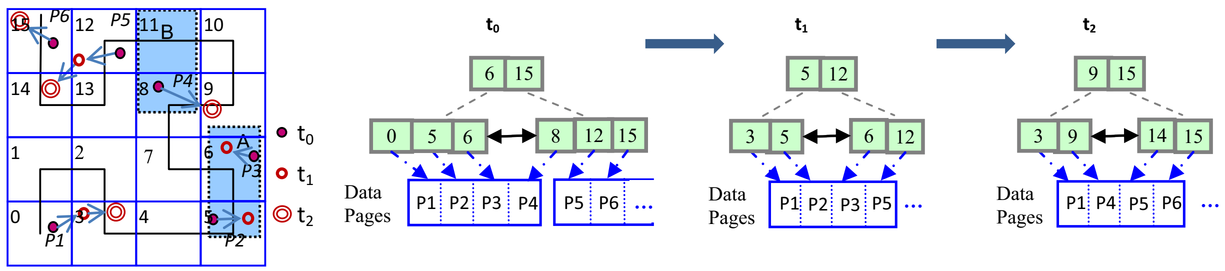

Fig. 3 An Example of Moving Objects and the In-memory Components of DIME with B+-trees at t_0, t_1, and t_2

A straightforward approach to answer continuous queries is to process these queries as range queries periodically. However, it is not feasible when the number of continuous queries is large. Several approaches based on R-trees or hash tables have been proposed to process the continuous moving queries over moving objects by indexing both objects and queries. SINA [15] manages objects and queries by using hashing techniques, and incrementally processes positive and negative updates. Another approach, MAI [16], 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 [17] has been proposed to optimize the communication and query reevaluation costs due to frequent location updates. Recently, concurrent continuous query processing on the R-tree [18] and linear spatial indices [19] have been proposed to support concurrent operations for moving object management.

The disposable index framework for moving objects proposed in this paper is applicable to both the R-tree family and the linear spatial indices. It is capable of reducing the I/O costs of location updates and providing a hybrid storage model on these indices.

III. Preliminary

Before presenting the construction of DIME and the corresponding query processing algorithms, we introduce the overall design of the proposed framework.

A. Terms and Assumptions

In this framework, as illustrated in Fig. 4, a new component of spatial index is constructed after every period Δt, (namely, *phase*). Δt is defined as $\Delta t_{mn}/n$, where Δt_{mn} is the maximum time interval for any moving object to report its new location, and n is the number of phases in each Δt_{mn}. Each component is an independent spatial index to index the locations updated in a corresponding period Δt, with lifetime $(n+1)*\Delta t$. During its lifetime, an indexing component can be traversed to answer spatial queries, but does not respond to any delete operations. A component only resides in memory and handles insert operations during its first phase. After the first phase, it is moved to the secondary storage. After being initiated for $(n+1)$ phases, based on the definition of Δt_{mn}, a component only contains obsolete locations, and thus is entirely disposed. In case that there is an object has not been updated before disposal, it shall be inserted into the constructing component based on the old location and velocity. Note that Δt can be adjusted according to application scenarios to ensure the max number of updates in a phase can fit in memory. The above concepts are summarized in TABLE I.

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.

Window query: Each query window is represented as a spatial box; each query submits its query window to the database once.

Continuous query: Each continuous query is represented as a moving window; each query periodically refreshes its new query window.

In addition, we assume that the moving objects log their last report timestamp and send it along with the new report. Thus update operations can easily locate the obsolete locations in the proposed index framework.

TABLE I. Terms and Notations

Concept	Expression	Description
Maximum time interval	Δt_{mn}	Maximum time interval for moving objects to update locations
Phase	$\Delta t = \Delta t_{mn}/n$	Time interval to construct an indexing component
Component	C_t	Indexing component constructed by timestamp t
Lifetime	$Lt = (n+1)*\Delta t$	Time period from constructing an indexing component to disposing it

B. Framework

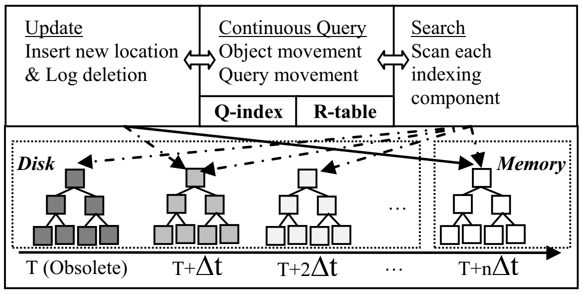

Fig. 4 Framework of DIME

A moving object access framework for DIME is designed as follows. This framework consists of a set of indexing components for snapshot query processing, and two auxiliary indices for continuous query processing, as shown in Fig. 4. The supported spatial operations are location update, window search, object movement, and query movement. Location update takes object ID, velocity, last update time, current time, and new location as inputs, and updates the object index. Window search takes query ID as input, and outputs the set of objects covered by the query window. Object/query movement (occurs in continuous query processing) updates location on

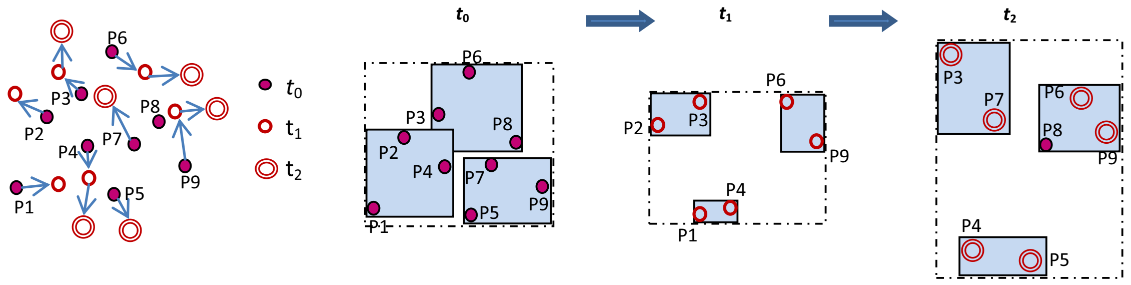

Fig. 5 An Example of Moving Objects and the In-memory Components of DIME with R-trees at t_0, t_1, and t_2

object/query index, searches on query/object index, and refreshes continuous query results.

DIME is a generic framework, as most spatial indexing structures can be applied to construct the indexing components. To better demonstrate the design and the algorithms, we apply the R*-tree and a linear spatial index (B+-tree integrated with SFC) correspondingly in this section. The Hilbert Space-Filling curve [20], which preserves the spatial proximity of objects, is applied to divide the space into non-overlapped cells, and map each object into a particular cell and each query window into a set of corresponding cells. Thus the cell IDs of moving objects can then be indexed by a B+-tree. Each entry in the leaf nodes of the B+-tree points to the data page that stores the objects in its corresponding cell.

To process the continuous queries over moving objects with efficiency and scalability, an additional index (*Q-index*) is applied to index current locations of queries, and another hash table (*R-table*) stores the query results. To collaborate with DIME based on the R*-tree, *Q-index* can be implemented as another R-tree. In case the B+-tree and SFC are used in DIME, the cell IDs of moving queries can be indexed in *Q-index* by a hash table, where the cell IDs are hash keys and the pointers to the corresponding queries are the contents in each bucket.

C. Illustrative Examples

Examples of moving object dataset indexed by DIME are illustrated in Fig. 3 and Fig. 5, utilizing the B+-tree with SFC and the R-tree respectively. In these examples, object locations at three timestamps, t_0, t_1, and t_2 are included. Note that $t_1 = t_0 + \Delta t$, and $t_2 = t_1 + \Delta t$. For each timestamp, a B+-tree (in Fig. 3) or an R-tree (in Fig. 5) is constructed in memory to index only the objects that report new locations during Δt.

In Fig. 3, the whole 2-dimensional space is divided into equal-sized square cells associated with 1-dimensional IDs using a Hilbert curve of order 3. The B+-tree for t_0 is constructed in memory as the initial indexing component. In this initial component, all the existing moving objects at timestamp t_0 are indexed based on their SFC cell IDs. Any object that updates its location between t_0 and t_1 has its new location indexed in the t_1 component, and marks the old location as obsolete in the log for the component t_0. Similarly, the B+-tree for t_2 is the indexing component for all the moving objects that update locations between t_1 and t_2. In case the maximum time interval contains 2 phases, the indexing components for t_1 and t_2 are expected to cover all the moving objects. When the indexing component for t_2 is completely constructed, the indexing component for t_0 can be disposed. Therefore, at any time, DIME contains *n* read-only constructed indexing components and one in-memory constructing indexing component.

In Fig. 5, initially an in-memory R-tree with 3 leaf nodes covers all the 9 objects at t_0. There are 6 location updates between t_0 and t_1, and a two-layer R-tree is built for these new locations. Similarly, another R-tree is constructed for the locations updated between t_1 and t_2. The tree for t_0 is copied to secondary storage at t_1. In this example, assuming *n* equals to 2, the object *P8* has not been updated for $n*\Delta t$ at t_2. Therefore, the new location for *P8* is estimated based on its old location and velocity (assuming is 0) at t_0, and is inserted into the R-tree for t_2. After this update, the indexing components for t_1 and t_2 cover all the moving objects, and the R-tree for t_0 can be entirely removed.

For location update in DIME, each indexing component does not change after being completely constructed. A location update calculates its new location based on the timestamp of the in-memory constructing indexing component. Then it inserts the new location into that component, and flags the deletion of its old location in a constructed component. If an object does not update after the maximum time interval Δt_{mn}(e.g., *P8* in Fig. 5), the system will need to estimate its new location and insert it into the current constructing component. On the other hand, a window query needs to traverse all these components to retrieve the qualified objects, since the whole set of moving objects are covered by *n* indexing components. In these traversals, the query windows are revised based on the object velocities and component timestamps. Detailed algorithms for update and query processing are presented in Section IV.

Q-index

Cell:	5	6	8	9	11
Query:	A	A	B	A	B

R-table

Query:	A	B
Object:	P2, P3	P4

Fig. 6 Q-index for queries and R-table for results in Fig. 3 at t_0

For scalable continuous query processing, an example of *Q-index* for query index corresponding to queries in Fig. 3 is shown in Fig. 6. Because the objects in this example are indexed in linear spatial index, a hash table can be used to index the query location. In the *Q-index*, each cell covered by any query has an entry. These entries consist of the corresponding queries, and can be efficiently accessed by a given cell ID. In case the objects are indexed using the R-tree, as demonstrated in Fig. 5, the *Q-index* can be implemented as an R-tree. In addition to these indices, another hash table, *R-table*, is used to store the query results in memory. In the *R-table*, the query IDs are used as hash keys, and each entry stores a list of objects covered by a particular query. Fig. 6 shows an example of the *R-table* for query results. 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.

For continuous query processing in DIME, an object movement will update its location in the disposable indexing structure, then search the *Q-index* for affected queries, and finally refresh the corresponding query results in the *R-table*. A query movement needs to search all the indexing components for the objects covered by its new query range, update the range in the *Q-index*, and refresh its query results in the *R-table*. A query report operation simply visits the entry in the *R-table* and outputs its object list.

IV. Disposable Index Operations

Operations on disposable index cover both snapshot and continuous queries. Snapshot query processing introduces location update and window search; continuous query processing requires object movement, query movement, and query report. Generic algorithms are presented in this section, and a specific design for the B+-tree-based DIME is discussed to illustrate how this framework works.

A. Snapshot Query Processing

Fundamental operations for snapshot query processing include location update and window search. The algorithms of these two operations are presented as follows.

1) Location Update: The operation of location update takes the object ID, last update time, current update time, new velocity, new location of the moving object, and the disposable index as input parameters. Involved structures include the disposable index, and a log buffer associated with each indexing component. This operation inserts the new location into the current indexing component, and reports the deletion of the old location to the indexing component corresponding to the last update time. Algorithm 1 presents the two phases of location update on disposable index.

```
Algorithm Location_Update
Input: Oid: Object ID, TS_old: Last Update Timestamp, TS_new: New
       Update Timestamp, loc_new: New Location of Object, vel_new:
       New Velocity of the Object, DI: Disposable Index of Objects
Output: Nil

//Phase1: Insertion
loc_cur = (DI[now].TS – TS_new) * vel_new + loc_new;
DI[now].insert(loc_cur); //update current component to insert loc_cur

//Phase2: Deletion
DI[TO] = DI.find(TS_old); //find the component for object's last update
If (DI[TO]!=DI[now])
    DI[TO].flag_obsolete(Oid); //flag the deletion of Oid
Else
    DI[now].flag_obsolete(Oid, loc_cur); //flag the valid location
Return;
```

Algorithm 1. Location_Update

Phase 1 -- Insertion. In this phase, the algorithm first calculates the current location of an object based on its velocity and reported location. The location to be inserted is computed as *(Current Component's Timestamp – reported Timestamp) * reported Velocity + reported Location*. Thus, all the moving objects indexed in one component have normalized locations based on the timestamp of the tree. In case that B+-trees are adopted, this algorithm calculates the SFC cell that the normalized location falls in, and uses the cell ID as the key to insert it into the current B+-tree.

Phase 2 -- Deletion. The algorithm identifies the component that has indexed the previous location of the object to be deleted based on its last report timestamp. After that, a log is added to that component indicating the deletion of this object. In case the previous report timestamp corresponds to the current component, a log will be added to record the object's valid location on the current component. This only happens on objects that update more than once within a phase. With these logs, there is no need for an actual deletion. When using the B+-trees in this framework, the deletion information is added to the associated log, and no tree modification is needed.

In the example in Fig. 3, the object in cell *0* at t_0 reports its current location *l* and velocity *v* at *ts*, where $t_0<ts<t_1$. The system calculates its new location at t_1 as $l_{t1} = l + v * (t_1 – ts)$, which is mapped by the Hilbert Curve to cell *3*. Then l_{t1} is inserted into the leaf node for cell *3* in the current B+-tree for t_1. In phase 2, a log is created to indicate that this object for t_0 is obsolete.

2) Window Search: The window search operation takes the current search time, a rectangular range, and the disposable index as inputs, and outputs the objects that fall in the range at the search time. This operation traverses each existing component in the disposable index to retrieve objects, validates these objects, and returns them as results. The detailed process is described in Algorithm 2.

```
Algorithm Window_Search
Input: TS_Q: Query Timestamp, Win: Query Window, DI: Disposable
       Index of Objects
Output: Res: Set of Objects

//Phase1: Parallel Search
For (int i = 0;  i < DI.count; i++)
        Win[i] = DI[i].adjust (Win); //adjust Win according to the maximum
                   velocity of indexed objects in DI[i]
        Res[i] = DI[i].search(Win[i]); //retrieve the objects covered by Win[i]
        Res[i] = Res[i] – DI[i].obsolete_set; // remove the objects marked as
                   obsolete in DI[i]

//Phase2: Union
Res = $\bigcap_i \mathrm{Res}[i]$;
Return Res;
```

Algorithm 2. Window_Search

Phase 1 -- Parallel Search. In this phase, the algorithm handles the window search on each component in the disposable index. For each component, the search window needs to be adjusted using the maximum velocity to multiply the difference between the search time and index timestamp. This window adjustment is similar to the corresponding approach in the B^x-tree [11]. After retrieving the objects using revised windows, for each component, the algorithm refines the object set by adjusting their locations using each individual velocity. Then the object set for each component is validated against the obsolete logs to remove invalid objects, and is returned. This phase can be executed on a parallel computing structure to achieve optimal performance. When adopting the B+-trees in DIME, this search operation needs to convert adjusted query windows into intersected SFC cells. Then all the existing B+-trees are traversed following the B+-tree search operation, and their associated logs are checked to return the query results.

Phase 2 -- Union. In this phase, the search operation generates the final result set as the union of the result sets from each component.

For instance, assuming Δt_{mn} contains two phases, in the example in Fig. 3, a window query issued at t_2 needs to traverse both the B+-trees for t_1 and t_2. Its original query window is applied on the B+-tree for t_2. A revised window, enlarged with the maximum object velocity at t_1, is applied on the B+-tree of t_1. The results from the B+-tree for t_1 are then validated against the obsolete logs of the tree to remove deleted locations, and combine with the results from the B+-tree for t_2 as the final result set.

B. Continuous Query Processing

Utilizing the snapshot query processing as the fundamental, continuous query processing can be designed by utilizing the *Q-index* and *R-table*.

1) Object Movement: The operation of object movement takes the object ID, last update time, current update time, new velocity, new location of the moving object, as well as the disposable index, *Q-index* and *R-table*, as input parameters. This operation updates the object location on the indexing components, identifies the queries that cover either the old location or new location from the *Q-index*, 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.

Phase 1 -- Object Location Update. This operation performs a location update (Algorithm 1) for the object. It inserts the adjusted new location into the current component. Then a log for deletion is added to the indexing component that corresponds to the last update timestamp.

Phase 2 -- Query Search. It retrieves the queries that cover the new location or old location of the object by looking up the *Q-index*. The affected queries are retrieved by computing their topological relation with the adjusted locations. In the B+-tree-based DIME, the *Q-index* can be implemented as a hash table as shown in Fig. 6. While the algorithm searches for the intersected queries, the corresponding SFC cell IDs are used as the keys to retrieve the queries. These queries are then further verified by comparing to the new location or old location of the moving object.

Phase 3 -- Result Refresh. The algorithm 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 out of, and is added into the entries for the queries that the object is moving into.

2) Query Movement: The proposed operation of query movement updates the location of the given query in the *Q-index*, 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 disposable index as input parameters. This operation consists of three phases, object window search, query location update, and result refresh.

Phase 1 -- Object Window Query. The algorithm traverses all the existing indexing components to locate the objects covered by the new query window, utilizing the window_search algorithm (Algorithm 2). A set of objects are retrieved at the end of this phase

Phase 2 -- Query Location Update. The algorithm updates the corresponding entries of the *Q-index* by deleting the old query window and inserting the new window. In case that B+-trees are adopted for indexing objects, this algorithm converts the query locations into intersected SFC cells, and uses these cells as keys to update the corresponding entries in the hash-based *Q-index*.

Phase 3 -- Result Refresh. The results of the given query in the *R-table* are replaced with the objects retrieved in Phase 1. Thus the *R-table* can correctly reflect the current query locations and object locations.

V. Performance

DIME reduces the location update cost by eliminating the index modification for delete operations. As a popular benchmark for spatio-temporal indices, the B^x-tree, which constructs a new sub-tree for each phase, has demonstrated outstanding performance for moving object management. Therefore, we analyze the performance of B+-tree-based DIME by comparing against the B^x-tree, in terms of I/O cost and space cost.

A. I/O Cost

Assuming the cost of reading a disk page is *DR*, writing a disk page is *DW*, traversing a disk-based B+-tree is *DT*, and modifying this tree for insert/delete is *DM*, a typical insert/delete operation costs *DR+DW+DT+DM* on a B^x-tree. Given the size of moving object set *N*, both *DT* and *DM* are proportional to *logN*. Since DIME does not require modifying the B+-tree for delete operation, it only needs to log the deleted object ID in memory, whose cost is a minor constant. To move an in-memory component to secondary storage, DIME needs constant I/O cost which is proportional to *logN*. In order to dispose an obsolete indexing component, DIME requires one I/O to mark the deletion of that B+-tree file and another disk I/O to delete the corresponding data file. Therefore, the I/O cost for disposing a component on DIME is constant. The overall location update cost on a B^x-tree, including an insertion and a deletion, can be calculated using the following expression: 2*(*DR+DW+DT+DM*). On the other hand, without the hybrid storage design, the location update cost on DIME is only *DR+ DW+DT+DM*, which is half to that on a B^x-tree. With the hybrid storage design, the disk I/O for update becomes memory operation except for copying a component to and disposing a component from disk.

The costs for a search operation on both the B^x-tree and DIME are similar, because they use similar indexing structures and both create new parts for every Δt. The B^x-tree and DIME both need to issue one adjusted query for each Δt to accomplish a search operation. Although DIME requires validating the search results by checking the deletion log in memory, its cost can be neglected comparing to disk I/O operations. Applying the above notations, the search cost on the B^x-tree can be represented as $DT*(n+1) + DR*l$, assuming l data pages are retrieved on average. A search on DIME is $DT*n + DR*l$, because DIME has one component in memory and its traversal cost is negligible. Practically, the B^x-tree may contain less number of objects than DIME because it physically deletes obsolete objects. However, the components in DIME are often one level shorter than the B^x-tree, which compensates the overhead from the slightly larger total size.

B. Space Cost

Similarly to the B^x-tree, DIME contains multiple components that correspond to different time periods. Because DIME does not delete the obsolete locations from the index trees, it requires more space than the B^x-tree. Assume there are *n* phases in both DIME and the B^x-tree, and *k* objects update their locations in each phase, where $k*n \geqslant N$. Because the size of a B+-tree is proportional to the number of keys, it can be denoted as $p*k$, where *p* indicates the size factor. Utilizing the above notations, the size of DIME can be calculated. Based on the design of the disposable index, there are $n+1$ B+-trees at the same time. In those trees, *n* are fully constructed on disk and one is being constructed in memory. In DIME, main memory buffer is used to log the obsolete objects. Since only the object IDs are stored in this buffer, the memory cost for logging the obsolete objects is *size_of(object ID) * (# of objects in DIME - # of objects in dataset)*. Thus, the upper bound of the buffer size is *size_of(object ID) * ((n+1)*k - N)*. Therefore, the total size of this index structure is $n*p*k$ on disk and at most $p*k$ + *size_of(object ID)* ((n+1)*k - N)* in memory.

In the B^x-tree, the number of indexed objects is constant if the object set is fixed. Therefore, the total size of this B^x-tree is $p*N$ on disk.

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 I/O cost (average number of disk page accessed) of operations on moving objects. The benchmark data sets were generated by a network-based moving objects generator [21] using the road network of City of Oldenburg. 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. Meanwhile, the object movements simulated by the generator were translated into object location updates (and query updates for continuous query). Both B+-tree-based and R*-tree-based DIMEs were conducted. The B+-trees were constructed as indexing components based on the moving object set and the Hilbert curve with different orders. The R*-trees were adopted as components of R*-tree-based DIME.

For comparison, the query processing based on the B^x-tree [11] was implemented. The B^x-tree creates a new sub-tree in each phase, processes location updates by inserting new locations and deleting old locations on different sub-trees. The search operation on the B^x-tree traverses each sub-tree to find the objects. We use Bx to represent the B^x-tree approach, and DI for the B+-tree-based DIME, with a followed number to indicate the order of the Hilbert Curve. Meanwhile, comparisons between the R*-tree-based DIME and the memory-based R*-tree were conducted based on throughputs. DIME based on the R*-trees is indicated as DI_R. The memory-based R*-tree only indexes spatial information, and it was used to keep the current locations.

In the experiments, five parameters were varied to simulate different application scenarios and demonstrate their respective impacts on the system performance. These parameters are listed in TABLE II.

The proposed framework was implemented using JDK 1.5. The experiment system was built on a desktop with a Pentium-D 2.8 GHz CPU and 1 GB memory assigned to JVM. When comparing DI to Bx, average page accesses of ten rounds of executions were counted without distinguishing disk and memory operations, in order to demonstrate the performance of DIME before taking advantage of in-memory processing. Note that a higher order of Hilbert curve results in a larger tree but less location comparisons during retrieval. To illustrate the efficiency of hybrid storage and disposable components, average throughputs of ten rounds were collected when comparing DI_R to the memory-based R*-tree. Each round of execution contains *(Mobility + Q_ratio * Mobility)* operations. Each page was assigned 4 KB, and 10 pages were used as buffer. The *Q-index* and *R-table* for continuous query processing were stored in memory. A set of 300K initial moving objects and moving queries were used in the experiments.

TABLE II. Experiment Parameters

Parameter	**Description**	**Range**	**Default**
Order	The order of the Hilbert curve applied. It determines the number of cells for the whole space.	8~10	8
Mobility	The number of location updates for objects issued in a phase.	25K~150K	100K
Q_size	The side length of query window for each query (600 is about 2%). It simulates query ranges in different applications.	100~600	100
Phase	Number of indexing components constructed in Δt.	2~7	2
Q_ratio	The number of queries (query location updates for continuous query) compared to Mobility.	5%~30%	25%

A. Snapshot Query Processing

1) *Update I/O vs. Mobility:* In this set of experiments, the impact of mobility was studied by capturing the average number of page accesses of location updates with different component sizes. The experiment results on the B+-tree-based solutions are illustrated in Fig. 7, where the *X*-axis indicates the mobility value and the *Y*-axis represents the number of page acesses per location update. Basically, a higher mobility means more objects reporting their movements within one phase. Consequently, when more movements need to be processed, the size of each indexing component becomes larger. When the SFC order was 8, the update cost became lower because the heights of the trees were not changed, and a larger tree has less channce for re-organization. As shown in Fig. 7, when the mobility increased from 25K to 150K, the number of page accesses of DI on all the data sets decreased from 2.5 to 2.1. Interestingly, when the SFC order was 10, the I/O costs of DIME and the B^x-tree increased along with the mobility. Especially when the mobility changed from 25K to 100K, the average numbe of page accesses of Bx increased from 6 to 6.8, and the DI changed from 3 to 3.4. This was because the B+-

trees increased the heights when the mobility changed from 25K to 50K. When the mobility was 50K or 75K, the indexing trees were relatively sparse, therefore structure changes easily occurred.

Comparing the average number of page accesses between DIME and the B^x-tree, it is clear that the B^x-tree consumed twice of DIME, for both order 8 and order 10 SFCs. This result is consistent with the theoretical analysis in Section V. It's because that one update in DIME only needs to perform insertion, while an update requires insertion and deletion in the B^x-tree.

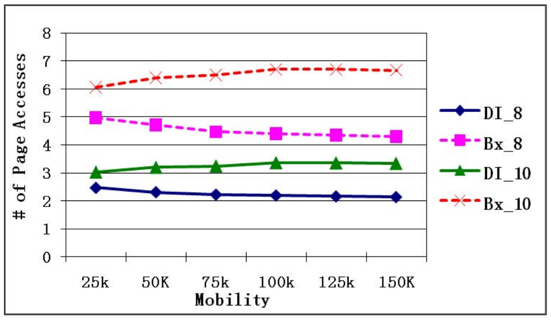

Fig. 7 Update I/O vs. Mobility on B+-tree-based DIME

On the other hand, Fig. 8 illustrates the throughputs of update operations on the memory-based R*-tree and R*-tree-based DIME. The update cost on the R*-tree was 2.6-2.8 times of the corresponding DIME. The major reason was that delete operations caused a significant amount of tree adjustments on the R*-tree. Both DIME and the R*-tree slightly decreased their throughputs when the mobility increased, because the index size was increased.

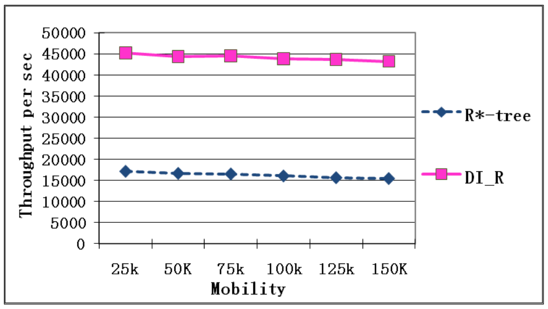

Fig. 8 Update I/O vs. Mobility on R*-tree-based DIME

2) Search I/O vs. Mobility: The mobility determines the size of each indexing component. This set of experiments demonstrated the impact of mobility on search cost. The results on the B+-tree-based solutions are visualized in Fig. 9, where the *X*-axis indicates the number of location updates in each phase and the *Y*-axis represents the average number of page acesses in each search operation. The mobility was increased from 25K to 150K in the experiments. When the order was 8, there was no significant performance change observed for DI and Bx as the mobility increased. Both DI_8 and Bx_8 spent about 4 page accesses regardless the change of the mobility. When the order of SFC was 10, a significant jump occurred between 25K and 50K. The average number of page accesses increased from 17 to 24 for both DI_10 and Bx_10, because the height of each indexing component was increased when the mobility reached 50K. After this jump, both DI_10 and Bx_10 had their number of page accesses stably between 24 and 25. In this set of experiments, the search performance of DI and Bx was very close. This observation matches the performance analysis in Section V.

Fig. 10 compares the search performance between the memory-based R*-tree and DIME. The memory-based R*-tree achieved 25% higher throughput than DIME on average, and decreased its throughput by 2/3 when the mobility increased from 25K to 150K. This was because DIME needed to access one in-memory component and one on-disk component to retrieve the results, although each component was about 1/2 of the corresponding R*-tree. Moreover, the increased mobility resulted in larger indexing components, which caused higher search costs.

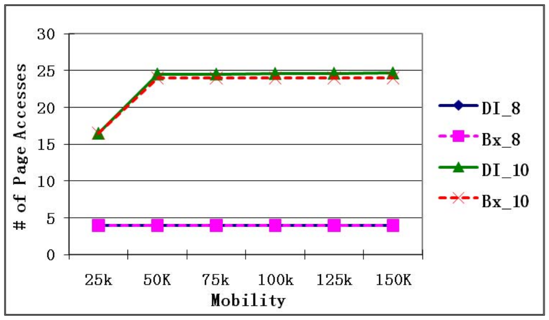

Fig. 9 Search I/O vs. Mobility on B+-tree-based DIME

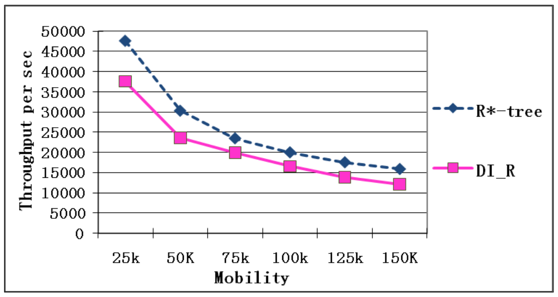

Fig. 10 Search I/O vs. Mobility on R*-tree-based DIME

3) Search I/O vs. Q_size: Performance of search operations against query size of DIME was studied in this set of experiments, as shown in Fig. 11 and Fig. 12. In these figures, the *X*-axis indicates the Q_size (side length of the query window) and the *Y*-axis represents the average number of page acesses in each search operation. When the Q_size increases, the query window covers more nodes in the indexing tree, therefore more pages need to be accessed. As shown in Fig. 11, when the Q_size increased from 100 to 600, the number of page accesses of both DI_8 and Bx_8 increased from 4 to 25. When the SFC order was set to 10, the average I/O costs of both DI_10 and Bx_10 increased quickly from 25 to about 130. The curves of order 10 rise faster than the curves of order 8 in the figure, because the B+-trees with order 10 contain more nodes, and the same area can cover more node in those trees than the B+-trees with order 8.

When the Q_size exceeded 300, the performance of DIME was slightly lower than that of B^x-tree. That was because the B^x-tree kept deleting the obsolete location from the subtrees, which decreased the size of the index. Thus a query window covered less nodes in the B^x-tree than in the disposable index. This affected the performance especially for large queries. When the query window was small, the impact of reduced size of the B^x-tree was compensated by the design of seperated B+-trees in disposable index. Concluded from this set of experiments, the B^x-tree and the disposable index performed similerly on search cost, and the B^x-tree slightly outperformed the disposable index on large query windows.

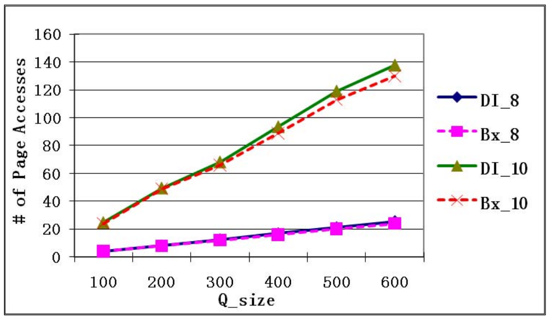

Fig. 11 Search I/O vs. Q_size on B+-tree-based DIME

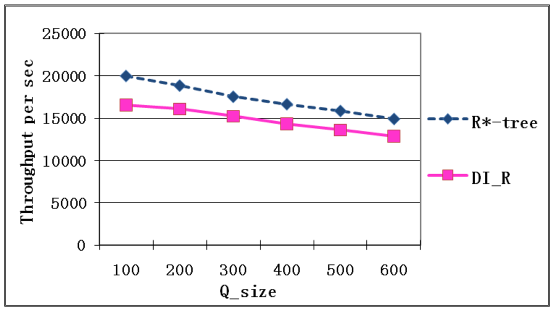

Fig. 12 Search I/O vs. Q_size on R*-tree-based DIME

Search performance of the R*-tree-based DIME and the memory-based R*-tree with regards to the Q_size is presented in Fig. 12. Both methods linearly decreased their search throughputs by about 25% when the Q-size increased from100 to 600. It can be observed that the search performance on both methods were sensitive to the Q_size.

4) Search I/O vs. Phases: The number of phases determines the number of components exist in DIME, and the number of subtrees in the B^x-tree. Since the R*-tree only captures a snapshot of moving objects, its performance is not affected by the number of phases, and was not discussed in this set of experiments. This set of experiments varied the number of phases from 2 to 7, and calculated the average number of page accesses in one search operation. The results are illustrated in Fig. 13, where the *X*-axis indicates the number of phases and the *Y*-axis represents the average number of page accesses in each search operation. Based on the search algorithm, all the indexing components need to be searched to acquire a complete result set. In this figure, both DI and B^x increased their I/O costs linearly when the number of phases increased. For the Hilbert Curve with order 8, the number of page accesses of both DI and Bx increased from 4 to 14. When applying the Hilbert Curve with order 10, the number of page accesses of both approaches increased from 25 to above 80.

It can be noticed that with order 10, the disposable index cost slightly less page accesses than the B^x-tree when it contained more than 6 phases. This is because each indexing component is designed as an independent B+-tree in the disposable index, rather than as a subtree of a large integrated tree in the B^x-tree. When the number of phases increased, this benefit became more significant.

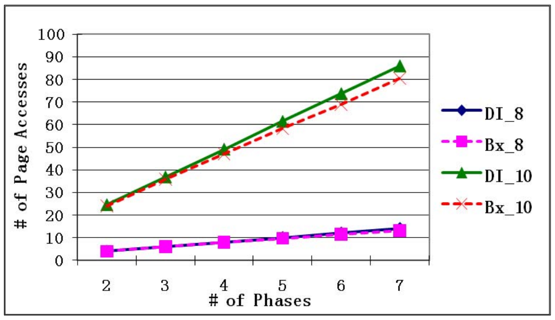

Fig. 13 Search I/O vs. Phases on B+-tree-based DIME

B. Continuous Query Processing

1) Movement I/O vs. Mobility: Fig. 14 and Fig. 15 demonstrate how the average number of page accesses in a movement changed with regards to the mobility. In these figures, the *X*-axis represents the mobility (from 25K to 150K), and the *Y*-axis indicates the average number of page accesses in a movement (object or query). Because a movement either contains a location update (in object movement) or a window search (in query movement), the shapes of the curves in Fig. 14 are actually the integrations of the curves in Fig. 7 and Fig. 9. As shown in the figure, the Bx_8 spent over 70% more page accesses than the DI_8, and the DI_10 cost 2.5 less page accesses than the Bx_10.

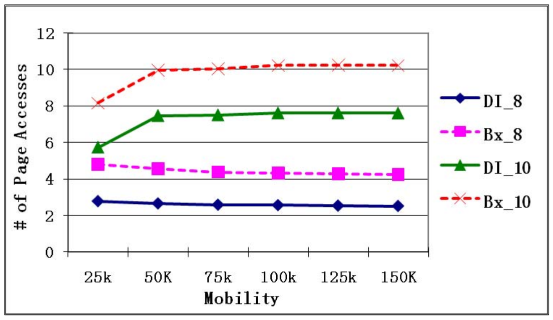

Fig. 14 Movement I/O Cost vs. Mobility on B+-tree-based DIME

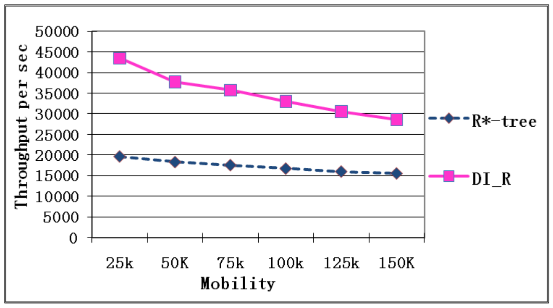

Fig. 15 Movement I/O Cost vs. Mobility on R*-tree-based DIME

Similarly, the average throughputs of the memory-based R*-tree and R*-tree-based DIME shown in Fig. 15 was the integration of update throughput and search throughput presented in Fig. 8 and Fig. 10 respectively. Since the update cost of the R*-tree was higher than its search cost, the movement cost of the R*-tree was dominated by its update cost. Similarly, the combined throughput of DIME was dominated by its search performance. As can be observed from the figure, the throughputs of DIME were 2.2-1.8 times of the corresponding memory-based R*-tree.

2) Movement I/O vs. Q_ratio: A larger Q_ratio indicates more query movements in one phase. Fig. 16 and Fig. 17 illustrate the impact of Q_ratio to the I/O cost per movement. In these figures, the *X*-axis represents the Q_ratio, and the *Y*-axis indicates the average number of page accesses in a movement (object or query). As shown in Fig. 16, when the Q_ratio increased from 5% to 30%, the DI and Bx with order 10 increased their I/O costs significantly by 4 page accesses. With order 8, both DI and Bx performed stably when the Q_ratio changed. Similar to Fig. 7, the DI_8 and DI_10 outperformed the Bx_8 and Bx_10 correspondingly. These results clearly demonstrated the optimized performance of the disposable index.

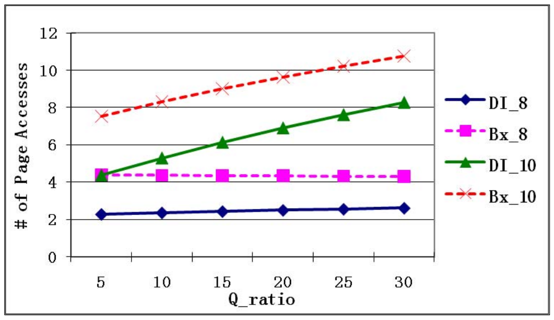

Fig. 16 Movement I/O Cost vs. Q_ratio on B+-tree-based DIME

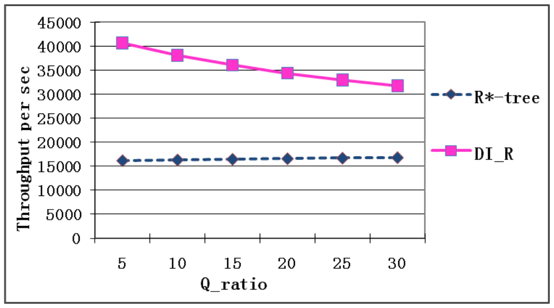

Fig. 17 Movement I/O Cost vs. Q_ratio on R*-tree-based DIME

As shown in Fig. 17, the memory-based R*-tree showed stable I/O cost when the Q_ratio increased from 5 to 30, because its update cost and search cost were comparable as can be observed from Fig. 8 and Fig. 10. The I/O cost of R*-tree-based DIME decreased from 41K to 32K when the Q-ratio increased. That was because the search operations on DIME cost much more than the updates.

In the above experiments, the performance of DIME confirms the theoretical analyses in Section V. The B+-tree-based DIME significantly outperformed the B^x-tree on location updates and continuous query processing, with similar search performance. On the other hand, the R*-tree-based DIME improved over 160% of the update performance compared to the memory-based R*-tree.

VII. Conclusions

This paper proposes a framework for moving object management with optimized update cost and independent indexing components. The proposed disposable index framework eliminates the deletion of obsolete locations on indexing trees and adopts hybrid storage to process the insertion of new locations in main memory. Most of the existing spatial indices can be adopted in the generic framework of DIME. Snapshot and continuous query operations are supported in DIME. Both the theoretical analysis and experimental evaluation have been conducted to validate that the proposed disposable index handles moving objects with outstanding efficiency and scalability. This work provides generic expandability of utilizing a variety of spatial indices, and offers hybrid storage and the ability of parallel searching.

References

[1] N. Beckmann, H. P. Kriegel, et al., "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," In *Proc. ACM SIGMOD,* 1990, p. 322-331.
[2] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," In *Proc. ACM SIGMOD*, 1984, p. 47-57.
[3] M. L. Lee, W. Hsu, et al., "Supporting Frequent Updates in R-Trees: A Bottom-Up Approach," In *Proc. VLDB,* 2003, p. 608-619.
[4] T. Sellis, N. Roussopoulos, et al., "The R+-tree: A Dynamic Index for Multi-dimensional Objects," In *Proc. VLDB*, 1987, p. 507-518.
[5] X. Xiong and W. G. Aref, "R-trees with Update Memos," In *Proc. IEEE ICDE*, 2006, p. 22-31.
[6] B. Lin and J. Su, "Handling Frequent Updates of Moving Objects," In *Proc. ACM CIKM* 2005, p. 493-500.
[7] L. Biveinis, S. Saltenis, et al., "Main-memory Operation Buffering for Efficient R-tree Update," In *Proc. VLDB*, 2006, p. 591-602.
[8] D. Sidlauskas, S. Saltenis, et al., "Trees or Grids? Indexing Moving Objects in Main Memory," In *Proc. ACM SIGSPATIAL GIS*, 2009, p. 236-245.
[9] J. Dittrich, L. Blunschi, et al., "Indexing Moving Objects Using Short-Lived Throwaway Indexes," In *Proc. the 11th International Symposium on Advances in Spatial and Temporal Databases*, 2009, p. 189-207.
[10] C. S. Jensen, D. Tielsytye, et al., "Robust B+-Tree-Based Indexing of Moving Objects," In *Proc. MDM*, 2006, p. 12-20.
[11] C. S. Jensen, D. Lin, et al., "Query and Update Efficient B+-Tree Based Indexing of Moving Objects," In *Proc. VLDB*, 2004, p. 768-779.
[12] M. L. Yiu, Y. Tao, et al., "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] D. Lin, C. S. Jensen, et al., "Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects," In *Proc. MDM*, 2005, p. 59-66.
[14] S. Chen, B. C. Ooi, et al., "ST^2B-tree: A Self-Tunable Spatio-Temporal B+-tree for Moving Objects," In *Proc. ACM SIGMOD*, 2008, p. 29-42.
[15] M. F. Mokbel, X. Xiong, et al., "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," In *Proc. ACM SIGMOD*, 2004, p. 321-330.
[16] B. Gedik, K.-L. Wu, et al., "Processing Moving Queries over Moving Objects Using Motion-Adaptive Indexes," *IEEE T Knowl Data En,* vol. 18, pp. 651-668, May 2006.
[17] H. Hu, J. Xu, et al., "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects," In *Proc. ACM SIGMOD*, 2005, p. 479-490.
[18] J. Dai and C.-T. Lu, "C3: Concurrency Control on Continuous Queries over Moving Objects," In *Proc. ICDE*, 2010, p. 121-132.
[19] J. Dai, C.-T. Lu, et al., "A Concurrency Control Protocol for Continuously Monitoring Moving Objects," In *Proc. MDM*, 2009, p. 132-141.
[20] H. Sagan, *Space Filling Curves*. Berlin, Germany. Springer, 1994.
[21] T. Brinkhoff, "A Framework for Generating Network- Based Moving Objects," *Geoinformatica,* vol. 6, pp. 153-180, Jun. 2002.