Efficient Concurrent Operations in Spatial Databases

Jing Dai

Dissertation submitted to the faculty of the Virginia
Polytechnic Institute and State University in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
In
Computer Science and Applications

Chang-Tien Lu, Chair
Ing-Ray Chen
Weiguo Fan
Edward A. Fox
Naren Ramakrishnan

September 4, 2009
Falls Church, Virginia, USA

Keywords: Spatial Database - Concurrency Control - Query Processing - Index

Efficient Concurrent Operations in Spatial Databases

Jing Dai

ABSTRACT

As demanded by applications such as GIS, CAD, ecology analysis, and space research, efficient spatial data access methods have attracted much research. Especially, moving object management and continuous spatial queries are becoming highlighted in the spatial database area. However, most of the existing spatial query processing approaches were designed for single-user environments, which may not ensure correctness and data consistency in multiple-user environments. This research focuses on designing efficient concurrent operations on spatial datasets.

Current multidimensional data access methods can be categorized into two types: 1) pure multidimensional indexing structures such as the R-tree family and grid file; 2) linear spatial access methods, represented by the Space-Filling Curve (SFC) combined with B-trees. Concurrency control protocols have been designed for some pure multidimensional indexing structures, but none of them is suitable for variants of R-trees with object clipping, which are efficient in searching. On the other hand, there is no concurrency control protocol designed for linear spatial indexing structures, where the one-dimensional concurrency control protocols cannot be directly applied. Furthermore, the recently designed query processing approaches for moving objects have not been protected by any efficient concurrency control protocols.

In this research, solutions for efficient concurrent access frameworks on both types of spatial indexing structures are provided, as well as for continuous query processing on moving objects, for multiple-user environments. These concurrent access frameworks can satisfy the concurrency control requirements, while providing outstanding performance for concurrent queries. Major contributions of this research include: (1) a new efficient spatial indexing approach with object clipping technique, ZR+-tree, that outperforms R-tree and R+-tree on searching; (2) a concurrency control protocol, GLIP, to provide high throughput and phantom update protection on spatial indexing with object clipping; (3) efficient concurrent operations for indices based on linear spatial access methods, which form up the CLAM protocol; (4) efficient concurrent continuous query processing on moving objects for both R-tree-based and linear spatial indexing frameworks; (5) a generic access framework, Disposable Index, for optimal location update and parallel search.

ACKNOWLEDGEMENT

First and foremost I offer my sincerest gratitude to my supervisor, Dr. Chang-Tien Lu, who has supported me throughout my Ph.D. research with patience and knowledge whilst allowing me the room to work in my own way. I attribute the level of my Ph.D. degree to his encouragement and effort. Without him this thesis, too, would not have been completed or written. One simply could not wish for a better or friendlier supervisor.

As committee members, Dr. Chen, Dr. Fan, Dr. Fox, and Dr. Ramakrishnan provided their precious suggestions and encouragement during each conversation we have had. Their unique views to the area I have been working on have revealed interesting issues and motivated enhancement to my research work.

In my daily work in the Northern Virginia Center, I have been blessed with a friendly and cheerful group of lab-mates. Yufeng not only helped me settle down and get used to the graduate life in VT, but also acted as a perfect model of hardworking researcher to me. Feng, Arnold, Ray, and Manu have supported me on my research topics through numerous extensive discussions. Weiping essentially taught me everything I know about UNIX. Meanwhile I appreciate the graceful help from all my friends that cheered my life.

Finally, I thank my family for supporting me throughout all my studies in U.S., especially my wife, Ying, for discussing my research, encouraging and soothing my soul, and taking care of our lovely son.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF ALGORITHMS

Chapter 1. INTRODUCTION

1.1 Motivation and Research Issues

In the last two decades, spatial data access methods have been proposed and developed to manage multi-dimensional databases as required in GIS, CAD, and scientific modeling and analysis applications. In order to apply the widely studied spatial access methods in real applications, concurrency control protocols are required for multi-user environments. Concurrency control for spatial access methods refers to the techniques to provide serializable operations in multi-user spatial databases. Specifically, the concurrent operations on spatial data should be safely executed and follow the ACID rules. The ACID rules are: Atomicity, Consistency, Isolation, and Durability. With concurrency control, multi-user spatial databases can process the search and update operations correctly without interfering with each other.

Concurrency control for spatial access methods is generally required in commercial multidimensional database management systems. These systems are designed to provide efficient and reliable spatial data management for business, government, and science areas. Usually they are required to handle a large number of simultaneous queries and updates reliably. In addition, concurrency control methods are required in many specific spatial applications that require fresh query results. For example, a taxi management system needs to find taxis close to the clients. Concurrency control methods need to be applied to isolate the taxi location updating and queries, so that the query results are consistent with an up-to-date snapshot of taxi locations. Another example could be a patrol vehicle to disseminate an alarm to nearby cell phones within a certain area. The database of cell phone locations has to be protected by a concurrency control protocol; otherwise the information may be sent to users outside of the area while some users inside the area could be missed.

Current commercial database systems usually support multiple isolation levels, including serializable isolation as the most protective level for concurrent operations. However, these serializable isolation implementations apply expensive locking strategies that are not optimized for spatial operations especially on moving objects. They either lock more resources, or lock resources longer, than necessary. Therefore, there are no efficient concurrency control protocols for serializable isolating spatial operations in these systems.

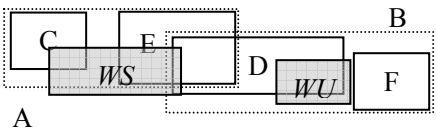
In order to apply the widely studied spatial access methods in real applications, concurrency control

protocols are required for multi-user environments. The simultaneous operations on spatial databases need to be treated as isolated operations without interfering with each other. In other words, the results of any operation have to reflect the current stable snapshot of the spatial database at commit time. Specifically, concurrency control for spatial access methods should assure the spatial operations are processed following the ACID rules. These rules are defined as follows.

- Atomicity - Either all or no operations are completed - in other words have the ability to undo if the operation fails.
- Consistency - All operations must leave the database in a consistent state.
- Isolation - Operations cannot interfere with each other.
- Durability - Successful operations must persist through crashes - in other words have the ability to redo committed operations to recover the database.

Current research on spatial concurrency control approaches mainly focuses on the consistency rule and the isolation rule.

Reflecting the ACID rules, protection against phantom update is used to measure the **effectiveness** of a concurrency control. Phantom update refers to such an update that occurs before the commitment of a search/deletion and falls in the range of that operation, while not reflected in the results of that search/deletion. An example of phantom update is illustrated in Figure 1, where objects *C*, *E*, *D*, and *F* are indexed in an R-tree, while leaf nodes *A* and *B* are their parents, respectively. *A* deletion with the window *WU* is done before the commitment of a range query with the window *WS*. Even though object *D* should have been deleted by *WU*, the range query returns the set {*C, E, D*}. That deletion is called a phantom update. A solution to avoid phantom update in this example is to lock the area affected by *WU* (which is *D* ∪ *WU*) to stop WS.

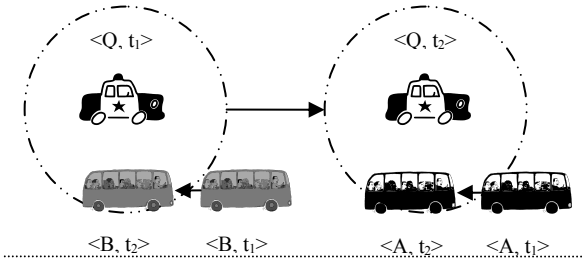


Selected objects by *WS*: *C, E, D.*

Figure 1. Example of Phantom Update.

In addition to phantom update, a lack of appropriate concurrency control protocols causes incorrect results in queries on moving objects. Figure 2 gives an example of inconsistent query results. In this example, a patrol vehicle *Q* keeps tracking all the buses within a given range of 0.5 mile. Timestamps t_1 and t_2 are two

consecutive query report times. *A* and *B* are two buses 1 mile away from each other, driving in the same direction towards the police vehicle. 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. The situation that returns the empty result set at t_2 is called **pseudo disappearance**, because bus *A* disappears in the result set of *Q* during its movement. This happens when *Q* is evaluated right after *A* deletes its old location, and the results of *Q* are reported before any other result refreshing related to *Q*. Bus *B* will be returned at time t_2 in a scenario called **back order**, where the query seems staying at its previous position while some objects have already updated their locations. This occurs when *B* has updated its new location in the result set of *Q*, and *Q*'s current results are reported before *Q* updates its new range. Back order also may result in an output {*A,B*} at t_2, where only *B* is back ordered and *A* is in normal status. Contrary 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 still output {*A*} as the result of *Q* at t_2, because in this situation, *Q* evaluates and outputs its results before the new location of *A* is updated in the database, and both $<A,t_1>$ and $<A,t_2>$ intersect with $<Q,t_2>$. All the above inconsistent scenarios can be avoided by a well designed concurrency control protocol for continuous query processing.



Correct result for Q at t_2: *A*

Possible results if without concurrency control:

Pseudo disappearance: on *A* or *A&B* → *Ø*, on *B* → *{A}*;

Back order: on *A* → *Ø*, on *B* → *{A,B}*, on *A&B* → *{B}*;

Pre-order: on *A* or *B* or *A&B*→*{A}*.

Figure 2. Inconsistency of Continuous Query w/o Concurrency Control.

The **efficiency** of concurrency control for spatial access methods is measured by the throughput of concurrent spatial operations. The throughput refers to the number of operations (i.e., search, insertion, and deletion) that are committed within a time unit. The issue to provide high throughput is to reduce unnecessary conflicts among the locks. For the example shown in Figure 3, although the update operation with window *WU* and the range query with window *WS* intersect with the same leaf node *A*, they will not

affect each other's results. Therefore, they should be allowed to access *A* simultaneously in a shared mode. Obviously, the finer lockable granules allow the most concurrency operations. However, that may require high overhead on lock maintenance. This is a tradeoff that should be considered when designing the concurrency control protocols. On the other hand, minimizing the lock durations can improve the throughput by releasing the resources as early as possible. Applying transaction management to handle concurrent operations usually requires the locks to be held until the commitment, which may harm the system performance by causing more conflicts during the unnecessary lock durations. To design an efficient concurrency control protocol, the lock durations have to be restricted to be as short as possible, and meanwhile fulfill the requirements for serializable isolation, consistency, and deadlock-freedom.

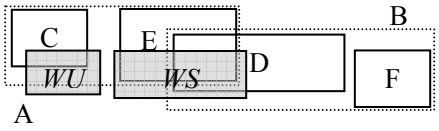


Figure 3. Example of Efficient Concurrency Control.

The concurrency control techniques for spatial databases have to be integrated with particular spatial access methods to process simultaneous operations. There are mainly two **categories** of spatial access methods, pure spatial access methods, and linear spatial access methods. The existing pure spatial data access methods, such as the R-tree family [1] and grid files [2], are efficient for querying spatial data items, especially on extended spatial objects, because of the spatial structure. These methods are quite different than the one-dimensional data access methods, e.g., B-tree and B+-tree. For instance, the R-tree groups spatial objects into Minimum Bounding Rectangles (MBR), and builds hierarchical tree structures to organize these MBRs. The grid file divides the whole data space into cells, and uses the cell map to access the corresponding data pages that store the data objects that intersect with particular cells. Another category of spatial data access methods is based on linear spatial indexing structures, represented by a Space-Filling Curve (SFC) [3] combined with B+-tree [4]. SFC splits the data space into equal-size rectangles and uses their particular curve *ID*s to index the objects in the cells on one-dimensional access methods, e.g., the B+-tree. Benefiting from the well-developed B-tree family, these methods are suitable for spatial point data and have low maintenance cost compared to the pure spatial access methods. However, the characteristics of spatial data and SFC mapping make it difficult to directly apply the traditional concurrency control protocols to these spatial access methods. In addition, to efficiently process continuous queries on moving objects, these spatial access methods have been modified to accommodate fast access to the continuous queries. Obviously, specific concurrency control protocols are required for these improved indexing structures to secure the correctness of the continuous queries.

Since the last decade of the 20th century, concurrency control protocols on spatial access methods have been proposed to meet the requirements of multi-user applications. So far these existing concurrency control protocols mainly focus on the R-tree family, and most of them were developed based on the concurrency protocols on the B-tree family. According to the lock strategy, the concurrency protocols can be roughly classified into two categories, link-based methods and lock-coupling methods. Because the spatial dataset is not usually globally ordered, the traditional link-based concurrency control techniques are difficult to adapt to spatial databases. Therefore the lock-coupling methods are popular to handle concurrent spatial operations.

Another potential solution is to adopt record-oriented transaction management techniques such as 2PL or multiversion approaches [5] on indices. The 2PL strategy tends to lock the resources until the commit point, which performs similarly to the sequential execution on indices. The multiversion approach requires a large number of different versions in frequent update scenarios, and thus leads to frequent undo/redo operations.

Transactional memory is a new technique for concurrency control. It integrates hardware support and software interface to provide a low level concurrency control so that the programmers can easily invoke the protocol rather than coding from scratch. The implementation of transactional memory is based on versioning transaction management. Generally, versioning approach is data record-based and expensive in frequent update scenarios. Since transactional memory has not been implemented as an applicable technique, it's difficult to determine how much the hardware can accelerate the versioning approach.

To design concurrent operations for spatial databases, there are four major **issues** to be considered, the lack of global order, the gap between indexing methods, the multi-component structure in indexing methods, and the efficiency of concurrent operations.

Lack of Global Order

Unlike one-dimensional data, spatial data items are not globally ordered. In other words, a spatial query may need to access multiple sets of consecutive data records. Although a pseudo global order, such as SFC, could be utilized to assign linear *ID*s to spatial data, these *ID*s cannot accurately reflect the spatial locality of the spatial data. The pseudo global order either increases the access cost, or translates a range query into multiple consecutive scans on an index structure. Therefore, efficient link-based concurrency control protocols for B-trees cannot be directly applied on spatial databases, and special concurrent operations need to be designed for spatial access methods.

Gap between Indexing Methods

The concurrency control protocols, which are not independent components in databases, have to be designed for the operations of indexing methods. Because of the differences among existing spatial indexing methods, it is difficult to develop a generic concurrency control protocol suitable for all the indexing methods. For example, the R-tree and R+-tree cannot share a single concurrency control protocol, since a point query may need to access multiple nodes on the same level of the R-tree, rather than a single node on the same level of the R+-tree. The gap between the R-tree and SFC, and the gap between the indices for stationary objects and moving objects, are even larger than that between the R-tree and R+-tree, so specific concurrency control protocols are required for each of them.

Multi-component Structure

Spatial access methods, especially those for moving objects, may contain multiple components. The spatial operations involving multiple indexing components need to be isolated from each other while accessing these components. However, the traditional concurrency control protocols provide protection only on single-component indices. So far, most of the research efforts have been dedicated to spatial query processing, and the design of corresponding concurrent operations is far behind. Since it is not trivial to design concurrent versions of operations on the multi-component indices, there is a lot to do to enable these access methods in multi-user environments.

Efficiency of Concurrent Operations

The operations on each spatial indexing method are originally designed for single-user environments. Some of them that originally perform quite well may greatly degrade the performance when directly integrated into concurrent access frameworks. For example, the nearest neighbor search on the R-tree [6, 7] will need to lock the whole range of the indexing tree while processing the search, thus no update operations could be performed in this period. In case that the spatial operations cannot match the efficiency requirement in multi-user environments, they have to be redesigned for the particular concurrency control protocols.

The accomplished research focuses on designing efficient and effective concurrent operations for spatial databases. As illustrated in Figure 4, according to the spatial indexing methods, two branches are followed to devise the concurrent access frameworks. One is for pure spatial indexing methods, represented by the R-tree family (left branch). An R-tree variant to enhance the search performance, ZR+-tree, is proposed as the indexing structure of the concurrent access framework. In addition, a lock-coupling-based concurrency control protocol, GLIP, is designed to support the concurrent operations on spatial indexing with object

clipping techniques, such as the R+-tree and ZR+-tree. Concurrent continuous query processing is then proposed by expanding the lock-coupling approach to multiple indexing components in an R-tree-based framework. The other (right) branch is linear spatial access methods. An efficient kNN search algorithm is designed for the access framework with SFC and the B+-tree. A Concurrent LocAtion Management protocol (CLAM) is proposed to support concurrent spatial operations on SFC with the B^{link}-tree. Concurrent operations, including range search and efficient location updates, are designed based on SFC and the B^{link}-tree. Concurrent continuous query processing is devised for monitoring moving objects in this framework. A generic framework, Disposable Index, is proposed to provide efficient location update, enable parallel search, and reduce concurrency control overhead on both spatial indexing trees and linear spatial indices. The research on these two branches is introduced in the following sections.

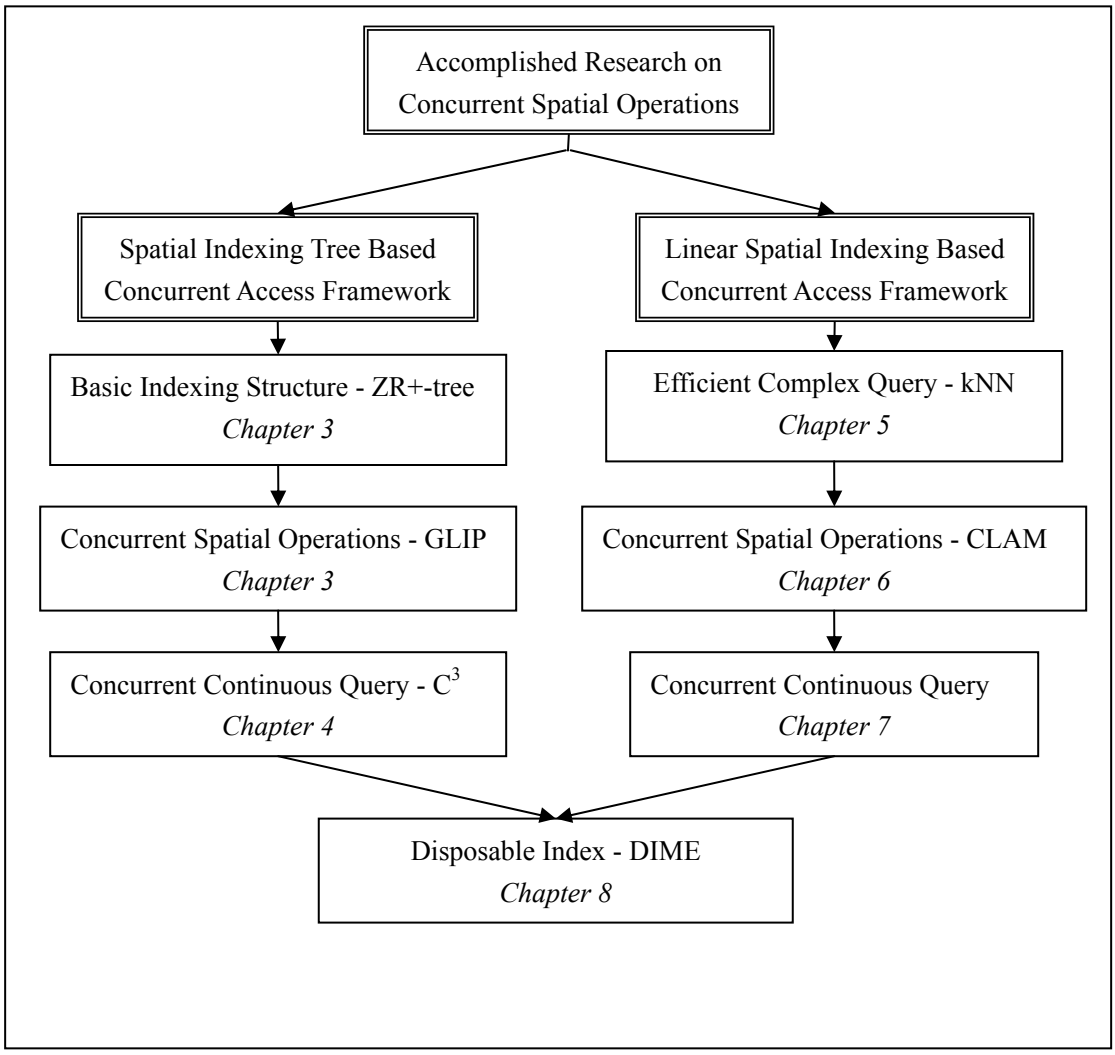


Figure 4. Overview of Research.

1.2 Concurrent Operations on Spatial Indexing Trees

Among the existing indexing structures, the R-tree family has attracted the most attention, as the tree structure is regarded as the most successful indexing structure for traditional databases. However, the current R-tree indexing has several limitations that prevent it from achieving better query performance. On the other hand, as an important issue related to indexing, concurrency control methods that support concurrent access in traditional databases are no longer adequate for today's multidimensional indexing structures due to the lack of a total order among the key values. In order to provide concurrency control in the R-tree structures, several approaches have been proposed, such as Partial Locking Coupling (PLC) [8], and granular locking approaches for the R-trees and GiSTs [9, 10].

In multidimensional indexing trees, overlapping of the nodes will tend to degrade the query performance, as a single point query may need to traverse multiple branches of the search tree if the point is in an overlapped area. The R+-tree [11] has been proposed based on a modification of the R-tree to avoid overlap between regions at the same level, using object clipping to ensure that point queries can follow only a single search path in the tree. The R+-tree exhibits a better search performance which makes it suitable for applications where search is the predominant operation and insertion and deletion are not common. For these applications, even a marginal improvement in search operations can result in significant benefits. Thus, the increased cost and complexity of updates is an acceptable price to pay in such cases. However, there are some limitations in the design of the R+-tree, such as its failure to insert and split nodes in some complex overlap or intersection cases [12], which will be discussed in Section 0. Furthermore, the R+-tree is not suitable for use with current concurrency control methods for multidimensional indexing trees, because in an R+-tree, a single object would be indexed in different leaf nodes. Special consideration measures need to be taken to support concurrent queries on R+-trees, while as far as we know, there is no concurrency control approach that directly supports an R+-tree.

We propose a Zero-overlap R+-tree, ZR+-tree, which compensates for this limitation of the original R+-tree in specific scenarios by eliminating overlaps of the leaf nodes. Furthermore, we have designed the concurrency control for the ZR+-tree, Granular Locking for clIPping indexing (GLIP), to provide phantom update protection, which also can be used in the R+-tree. The ZR+-tree, together with the proposed concurrency control, makes up an efficient and sound data access model for multidimensional databases.

On the other hand, access methods that consist of multiple spatial indices [13-15] have been proposed to process efficient continuous queries on moving objects. In these approaches, indices are constructed to

index both moving objects and moving queries, as well as a result set to maintain the query results. Although this is a practical design to efficiently answer continuous queries, none of the existing concurrency control protocols for spatial access methods can provide serializable isolation to the operations accessing multiple indices. Moreover, applying lazy update techniques on R-trees [16-19] to facilitate moving object management brings challenges to the task of concurrency control. An effective concurrency control protocol has to protect both the update buffer and the indices for correct results. Therefore, using R-trees to efficiently process continuous queries over moving objects requires concurrency control protocols to secure the consistency of the update buffer and indices, as well as to handle the multiple index components.

We propose a Concurrency Control protocol for Continuous queries (C^3) based on an efficient generalized access framework for continuous queries on moving objects. This generalized access framework maintains spatial indices for both objects and queries, and applies lazy update techniques for the whole framework to support frequent update.

1.3 Concurrent Operations on Linear Spatial Indexing Structures

Utilizing Space-Filling Curves (SFC) to linearly map multidimensional data to one-dimensional space, and retaining the usage of an one-dimensional index, could be a cost-efficient solution, because many well-developed modules in the traditional database management system can be reused, e.g., query optimization and storage organization. Comparing to the R-trees, those B-tree-based spatial access methods are inexpensive to handle modifications on indices, and particularly suitable for spatial point data. In this scenario, multidimensional access methods based on SFC have recently attracted many research efforts. Comparisons among different space-filling curves have been researched based on their characteristics [20, 21], and several query algorithms have been proposed to support the operations on SFCs [22-26].

Being devised for the traditional linear indexing structure, the linear spatial indices are not efficient to handle some complex spatial operations such as k-nearest neighbor (kNN) queries. Unlike the R-trees, simply expanding the scan range on the B-trees for a kNN search has to go through much more tree nodes than necessary. To achieve a better I/O cost, a kNN search may consist of a sequence of spatial range queries to gradually expand the search area. Therefore, an efficient design is needed to process kNN queries on this framework without significantly compromising on performance.

We propose an incremental kNN query processing method based on a multidimensional indexing structure with general SFCs. The characteristics of the grid space are utilized to design an efficient best-first search approach with optimized search range and simplified distance estimation. The proposed incremental kNN search is demonstrated to perform generally well on different SFCs. In addition, three general optimization techniques for spatial operations on SFC, namely multiple SFCs, query composition, and bitmap, are proposed to support efficient queries. These optimization approaches improve the performance by reducing the I/O cost of tree traversals.

Demanded by the promising frequently-operated spatial applications, concurrency control protocols should be designed to support spatial operations in multi-user environments. Among the concurrent operations on the B-trees, many concurrency control approaches have been proposed to make use of the global order of the entries in the leaf nodes of the tree [27-29]. These link-based approaches are proved to be efficient and easy to implement. Since the SFC based indexing methods use the B-trees as their one-dimensional indices, one intuitive idea is to apply the efficient link-based approaches to provide concurrency control. However, the SFC mapping cannot preserve the spatial locality completely, and the spatial operations require special considerations to assure the results are consistent with the dataset. For example, one spatial range query may contain several range queries on a B-tree, therefore it requires that all of these B-tree queries get the correct results reflecting the current version of the dataset at the commit point. To prevent these concurrent spatial queries from being interfered with by update operations, link-based approaches are not sufficient as they cannot protect multiple ranges, and therefore a new protocol is demanded for concurrency protection.

We propose Concurrent LocAtion Management (CLAM), a framework of concurrent location update and query based on the multidimensional indexing structure with general SFCs and the B^{link}-tree. This framework applies a concurrency control protocol which integrates the link-based approach with the lock-coupling mechanism. Spatial location update and range query algorithms based on the proposed protocol are designed.

Because of the efficient update and mature implementations, spatial access methods based on the B-tree are suitable for moving object management [30]. However, the continuous queries on moving objects bring two major 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 location updates and continuous queries, multiple indices can be applied to index objects, queries, and results accordingly [31]. The existing B-tree-based concurrency control techniques are not capable of providing

serializable isolation on operations involving multiple indices.

We propose a concurrency control protocol for concurrent spatial operations on moving objects based on B-tree and Space-Filling Curves (SFC). It supports concurrent continuous query processing involving multiple indices, and avoids pseudo disappearance, back order, and pre-order.

1.4 Generic Framework for Moving Object Management

Significant effort is made to remove the obsolete information and thus impacts the performance. As shown in the literature [17], the update I/O cost could be improved 44~68% by caching the delete operations. However, these cached delete operations still need to revise the index structure at certain time points. Further improvement can be expected if delete operations no longer change the index tree. On the other hand, existing moving object index approaches do not natively support parallel computing structures. Thus additional management (e.g., slice or copy) is needed to implement these indices in modern parallel structures. One location update will need to modify multiple slices or copies of the index, which increases the complexity and risks of inconsistency of the database system.

We propose an index framework, DIME (Disposable Index for Moving objEcts), to eliminate delete operations on the index structure and provide parallel query ability. Since the locations of moving objects are updated frequently, location information can become obsolete within a short time period. For this scenario, we provide a solution to conserve 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 removed without changing the internal structure. In the proposed solution, old locations of moving objects are disposed from the index without disk I/O cost. To address the parallel processing issue, we design a moving object index structure that is natively sliced based on the update timestamps of moving objects.

The basic idea is illustrated in Figure 5. The disposable index consists of multiple components (e.g., index trees) allocated corresponding to different time periods. Each component can be treated as an independent index for the objects updated in a corresponding time period. There are three operations supported on each component: **insert a location**, **search locations**, and **dispose the whole component**. A movement reported by an object triggers an insertion on the index, and flags its old location as obsolete. A search query traverses each component in parallel to identify the results. Taking the object movement in Figure 5 as an example, in the disposable index, the new location of o_8 will be inserted into a current 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, the disposable index requires no modifications on the index trees for deletion.

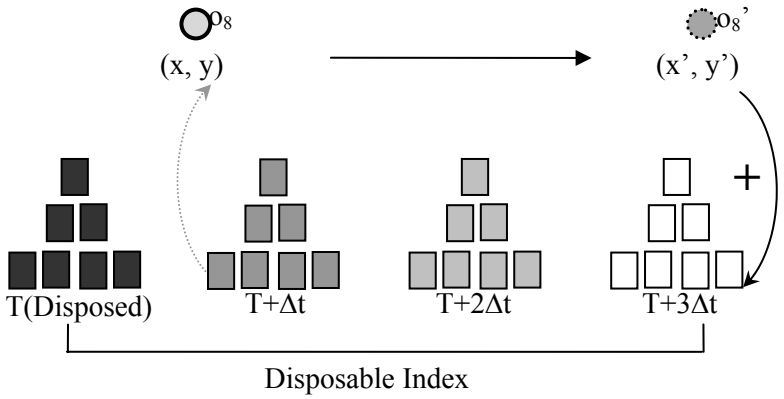


Figure 5. Location Update on Disposable Index.

The following design issues need to be resolved in this disposable index. 1) **Obsolete Location Handling**: how to efficiently record the obsolete locations without performing the actual deletion, and how to avoid these locations being returned in query results; 2) **Parallel Indexing Support**: how to construct independent index components, and how to set the number of components to balance the search performance and space cost.

The proposed moving object access framework is independent of the underlying index structure 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 the disposable index.

1.5 Contributions

This Ph.D. research developed concurrent spatial access frameworks for both stationary and moving objects. The developed solutions ensure serializable isolation, consistency, and deadlock-freedom on R-tree-based spatial indices and B-tree-based spatial indices. With these frameworks, spatial data management systems become more applicable in large scale multi-user environments. Specific contributions for each approach are listed as follows.

ZR+-tree and GLIP: The new multidimensional data access method, ZR+-tree, utilizes object clipping, optimized insertion, and re-insert approaches to refine the structure and remove limitations such as splitting

failures and inserting failures in constructing and updating R+-trees. The ZR+-tree provides better search performance than the R+-tree and the R-tree for both point queries and small range queries. Meanwhile, the proposed concurrency control protocol, GLIP, provides serializable isolation, consistency, and additional deadlock freedom on indexing trees with object clipping. ZR+-tree and GLIP form up an efficient and sound concurrent multidimensional data access model. This concurrent access framework is the first work that provides serializable isolation on the R+-tree family.

C^3: An efficient concurrent access framework for continuous query processing, C^3, is devised based on R-trees with a lazy update technique. Meanwhile, a sophisticated concurrency control protocol is designed to ensure serializable isolation, consistency, and deadlock-freedom on moving object management. The correctness of the proposed concurrent operations is proved, and the scalability and efficiency of the proposed framework are validated by the experiments on benchmark datasets. This concurrent access framework is the first work capable of handling concurrent continuous queries on moving objects indexed by the R-tree family.

Incremental kNN search on linear spatial index: The new kNN search performs a best-first search by utilizing the advantages of the access framework based on SFC and the B-tree. In addition, the three optimization techniques improve the search performance on linear spatial indices by reducing tree accesses. Experiments on real datasets have validated the scalability of the proposed incremental kNN query execution algorithm, as well as the improvement in performance achieved by each of the three optimizations.

CLAM: The new concurrent location management approach, CLAM, secures serializability and consistency without incurring additional deadlocks for concurrent spatial location updates and queries on multidimensional indexing structures with general SFCs and the B^{link}-tree. The concurrency control protocol of CLAM preserves the simplicity and efficiency of the link-based approach, and adopts the flexibility of the lock-coupling method. A formal proof is provided to show the correctness of the proposed CLAM framework. Experimental results on benchmark datasets validate the efficiency and scalability of the proposed concurrent operations and framework.

Concurrent continuous query processing on linear spatial index: A concurrent continuous query processing approach on a B-tree-based framework is developed. This concurrency control protocol fuses lock-coupling and link-based approaches to protect concurrent object updates, query updates, and continuous queries. It is proved that serializable isolation, data consistency, and deadlock-freedom are

guaranteed in the new framework. The scalability and efficiency of the concurrency control protocol is validated by an extensive set of experiments on benchmark datasets.

DIME: A generic index framework, Disposable Index for Moving objEcts, is developed with optimized I/O cost and parallel-ready structure. The new index structure has been extended to process continuous queries on moving objects. Theoretical analyses on the performance of the disposable index have been conducted from both I/O and space aspects. Extensive experiments on benchmark datasets validate the scalability and efficiency of the proposed index structure.

1.6 Organization of This Dissertation

The remainder of this Ph.D. dissertation is organized as follows. Chapter 2 reviews the related work of concurrency control protocols and spatial data management. Chapter 3 and 4 present the accomplished research on concurrent operations based on the R-tree family, including ZR+-tree, GLIP, and C^3. The accomplished results on concurrent access framework based on linear spatial indexing, including optimized kNN search, CLAM, and concurrent continuous query processing, are presented in Chapters 5, 6, and 7. The generic access framework, Disposable Index, is presented in Chapter 8. In Chapter 9, we conclude the research achievements of this dissertation, together with current publications and discussing future directions.

Chapter 2. RELATED WORK

This research is conducted based on the understanding of existing work on spatial access methods, spatio-temporal access methods, and concurrency control protocols. These related approaches are categorized into spatial indexing tree based and linear indexing based approaches, and reviewed in the rest of this chapter.

2.1 Spatial Indexing Tree Related Techniques

In this section, we first review the structure of the R-tree family, and discuss some limitations that affect R+-trees, as well as the techniques to adopt the R-trees for moving object management. Next, we discuss major concurrency control algorithms based on B-trees [32, 33] and R-trees that have been proposed over the last ten years. Finally, we will summarize the challenges involved in applying concurrency control to R-trees for both read-dominant applications and moving object management.

2.1.1 Indexing Structure

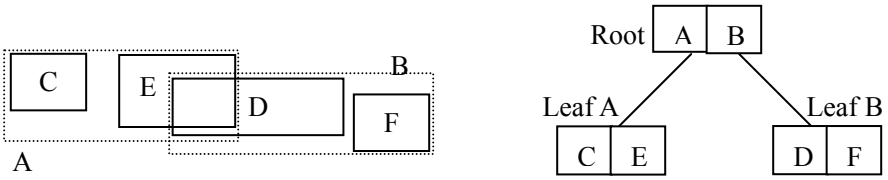
R-tree, an extension of the B-tree, is a hierarchical, height-balanced multi-dimensional indexing structure that guarantees that space utilization is above a certain value. In the R-tree, the root may have between 1 and M entries. Every other node, including leaf and internal nodes, has between m and M entries ($1 < m \leq M$). The leaf nodes hold references to the actual data and the Minimum Bounding Rectangle (MBR), which covers all the objects stored in the node. The internal node holds references that point to all its children (leaf nodes or the next level of internal nodes) and the MBRs corresponding to the children. Unlike B-trees, the keys in R-trees are multi-dimensional objects which are hard to define in a linear order. R-tree is one of the most popular multi-dimensional index structures as it provides a robust tradeoff between performance and implementation complexity [34]. Many variants based on the R-tree have been proposed to construct optimized indices [35], or manage spatio-temporal or high-dimensional data [36, 37]. In order to analyze the performance of the R-tree family, several evaluation approaches [38, 39] have recently been suggested. However, as the R-tree allows overlap in the same level nodes, in some cases the R-tree will have nodes with excessive space overlap and "dead-space," which decrease the performance in search operations, because for one search region there might be several MBRs in each level satisfying it at the same level. To optimize data retrieval performance, several variants have been proposed. For example, the R*-tree [40]

tries to minimize overlap for internal nodes and minimize the covered area for leaf nodes via forced-reinsert. The R+-tree also was defined to completely avoid overlap between nodes in the same level.

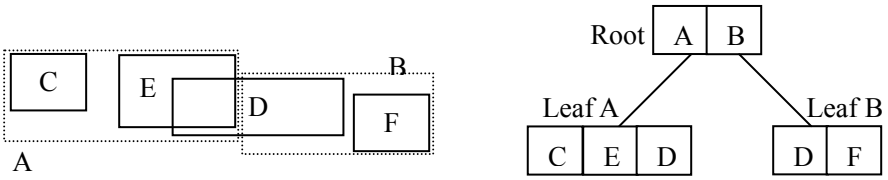
R+-tree was first proposed in [11]. R+-trees use a clipping approach to avoid overlap between regions at the same level [12]. As a result of this policy, point queries in R+-trees correspond to single path tree traversal from the root to a single leaf. For search windows that are completely covered by the MBR of a leaf node, R+-trees do guarantee that only a single search path will be followed, although this is not true in R-trees. Examples of the R-tree and the R+-tree are given in Figure 6 for comparison, where A and B are leaf nodes, and C, D, E and F are MBRs of objects. Because objects D and E overlap with each other in the data space, leaf nodes A and B have to overlap in the R-tree in order to contain the objects. In contrast, in the R+-tree, leaf nodes do not have to cover an entire object, so object D can be included in both leaf node A and leaf node B. As a result, R+-trees clearly have a better search performance compared to R-trees. Various experimental analyses of the relative performance of R-trees and R+-trees indicate that R+-trees generally perform better for search operations [34, 41]. This benefit however comes at the cost of higher complexity for insertions and deletions. The performance gain for search operations makes R+-trees ideally suited for large spatial databases where search is the predominant operation.

However, it is important to note the following limitations of the original definition and the operations of R+-trees. First, some objects may not be inserted into an existing tree, because of an extension conflict between several nodes on the same level. Figure 7(a) illustrates a 2-D example of this problem. In this case, when an object with MBR N is inserted into the tree, any one of nodes A, B, C and D could be chosen to extend to include the new object. The region N thus causes a deadlock in this scenario, because whichever node is chosen to include N will then overlap with another node. For instance, A will be touched if B is extended and B will be overlapped if C is enlarged. According to the definition and insertion algorithm used for R+-tree, none of these nodes is allowed to cover N while overlapping with other nodes. Therefore, the new object could not be inserted into the R+-tree. This issue was raised in [12], but no modified algorithms were presented to deal with it. Secondly, in some cases, it may not always be possible to split a node in a manner that satisfies all the properties of R+-trees. In an obvious case, a split is not possible when M+1 MBRs in a node with a capacity of M have the property such that the lower left corners (or upper right corners) of all the MBRs are the same. Figure 7(b) shows an example of this problem. Thirdly, the original R+-tree algorithm does not discuss how to clip an inserted object that overlaps with multiple untouched nodes. In this situation of insertion, the nodes that overlap with the object should be enlarged to cover the whole space of the object. As a matter of fact, observed from Figure 7(c), there could be multiple solutions to perform the node expansion, which lead to different tree structures. Among the two solutions in the

figure, solution b will generate a better indexing tree, because nodes A, B, and C cover less dead space than in solution a. The proposed ZR+-tree will be shown that offers a solution to these problems.

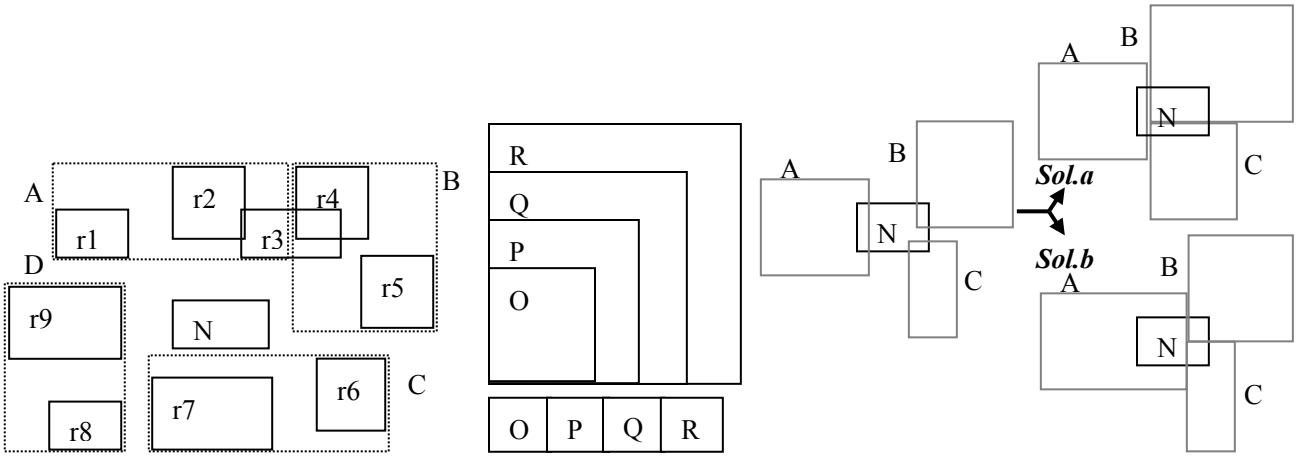


a) An example of R-tree



b) An example of R+-tree

Figure 6. Examples of the R-tree and R+-tree.



(a) Unable to Insert b) Unable to Split c) Different Solutions to Expand Nodes

Figure 7. Limitations in R+-trees.

Table 1. Techniques for Efficient R-tree Update.

	FUR-tree [16]	RUM-tree [17]	LGUR-tree [19]	R^R-tree [42]
Leaf Node Hashing	√			
Operation Buffer		√	√	
In-memory Tree				√

On the other hand, R-trees are usually considered as costly to update, which makes them unsuitable for moving object scenarios. Many techniques utilizing hashing and lazy update have been designed to reduce

the update cost of the R-tree or its variants. Table 1 lists several approaches from the literature for efficient update on R-trees and the corresponding techniques applied. The Frequent Update R-tree (FUR-tree) [16] processes the delete operation directly from the leaf nodes and simplifies the insert operation if the location change is small. This approach allows efficient location updates on R-trees, especially for small location updates, by eliminating the locate step in delete operations and in part of the insert operations. Lazy update approaches utilizing buffer memory have been proposed to reduce the I/O from another aspect. The R-tree with update memos, RUM-tree [17], applies main memory buffering to cache delete operations, so that they can be processed later when the particular leaves are accessed. Lazy group update on R-tree, LGUR-tree [19], caches not only the delete operation, but also the insert operation. Another approach, RR-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 [42].

In recent spatial database applications, a continuous query is a common type of query that keeps monitoring the moving objects in a certain area. One of the most challenging tasks in continuous query processing is to answer moving queries over moving objects. Several scalable approaches have been proposed to tackle this problem by indexing both objects and queries. SINA [13] indexes objects and queries using hashing techniques, and incrementally processes positive and negative updates. Another approach, MAI [14], constructs motion-sensitive indices for objects and queries by modeling their movements, so that some prediction queries can be supported. A generic framework for continuous queries on moving objects [15] has been proposed to optimize the communication and query re-evaluation due to frequent location updates.

2.1.2 Concurrency Control

Several concurrency control algorithms have been proposed to support concurrent operations on multi-dimensional index structures, and these can be categorized into link-based and lock-coupling based algorithms. The lock-coupling based algorithms [43, 44] release the lock on the current node only when the next node to be visited has been locked while processing search operations. During node splitting and MBR updating, these approaches have to hold multiple locks on several nodes simultaneously, which may deteriorate the throughput of the concurrency control.

The link-based algorithms [8, 29, 45-47] were proposed to reduce the number of locks required by lock-coupling based concurrency control algorithms. These methods lock one node most of the time during search operations, only employing lock-coupling when splitting a node or propagating MBR changes. For instance, the child node will hold a write-lock until a write-lock on the parent has been obtained. In some

cases, the lock on the child node may be retained during the processing [41], which may degrade the concurrency. The link-based approach requires that all nodes at the same level are linked together with right-links or bidirectional links. When a node splits, the newly created node will be inserted into the right-link chain directly to the right of the original node. All the nodes in the same chain are ordered by their highest key. In this way, a search operation can determine if there is a node split which has not yet affected the parent node by matching the highest key of the node to the one it is looking for. This method reaches high concurrency by using only one lock simultaneously in most operations on B-trees.

However, the linking approach can not be used directly in multi-dimensional data access methods as there is no linear ordering in multi-dimensional objects. To overcome this problem, a right-link style algorithm (R-link tree) [46] has been proposed to provide high concurrency control by assigning logical sequence numbers (LSNs) on R-trees. However, when a node splitting propagates and the MBR updates, this algorithm still applies lock coupling. Also, in this algorithm, additional storage is required by keeping extra information for the LSNs of associated child nodes. To solve the extra storage problem, Concurrency on Generalized Search Tree (CGiST) [47] applies a global sequence number, Node Sequence Number (NSN). The counter of NSN is incremented in a node split and the original node receives the new value, while the new sibling node receives the original node's prior NSN and its right-link pointer. In order for the algorithm to work correctly, multiple locks on two or more levels must be held by a single insert operation, which increases the blocking time for search operations.

Some mechanisms have been proposed that improve the concurrency based on the above linking techniques. A top-down index region modification (TDIM) technique [45] has been introduced to perform MBR updates combined with an insert operation, which traverses an index tree from the top. In addition, the MBR updates can be performed in piecemeal fashion without excluding search operations. Besides TDIM, optimized split algorithms such as Copy-based Concurrent Update (CCU) and CCU with Non-blocking Queries (CCUNQ) also have been proposed [45]. However, there are still limitations on delete operations, as this approach does not take into account the protection in multiple paths while deleting. For CCU, extra spaces are needed for splits, which increase the complexity. Partial lock coupling (PLC) [8] has been proposed to avoid query delays due to MBR updates for multi-dimensional index structures. The PLC technique increases concurrency by using lock coupling only in cases of MBR shrinking operations, which are less frequent than expansion operations. The major weakness of PLC is that an exclusive lock is held during the propagation, and the algorithm does not provide phantom update protection.

Phantom update refers to the update operation that occurs before the commitment and in the ranges of a

retrieval, but is not reflected in the retrieved results. Concurrent data access through multi-dimensional indexes introduces the problem of protecting the query range from phantom updates. The dynamic granular locking approach (DGL) has been proposed to provide phantom protection in R-trees [9] and GiST [10]. The DGL method dynamically partitions the embedded space into lockable granules that adapt to the distribution of the objects. The leaf nodes and external granules of the internal nodes are defined as the lockable granules. External granules are additional structures that partition the non-covered space in each internal node to provide protection. According to the principles of granular locking, each operation requests locks on enough granules such that any two conflicting operations will request conflicting locks on at least one granule in common. The DGL approach provides phantom update protection for multi-dimensional access methods and granular locks can be implemented more efficiently compared to predicate locks, but the complexity of DGL may impact the degree of concurrency because of its complexity.

2.1.3 Challenges of Applying Concurrency Control

Several efficient key value locking protocols to provide phantom protection in B-trees have been proposed [32, 48, 49]. However, they cannot be applied directly to multi-dimensional index structures such as R-trees, because for multi-dimensional data, a total ordering of the key values on which these protocols are based is not defined.

Granular locking protocols such as GL/R-tree [9, 10] for multidimensional indices have been proposed, but again none can be directly applied in the R+-tree. An example will show why the original GL/R-tree is not sufficient to provide phantom protection for the R+-tree. The GL/R-tree defines two types of lockable granules: leaf granules that correspond to the MBR for each leaf node and external granules that are defined as ext(internal node)=(MBR for the internal node)–(MBRs for each of its children). In Figure 8 (assuming A and B are leaf nodes), the search window *WS* requires shared locks to be placed on the lock granules A, whereas the update window *WU* requires exclusive locks to be placed on B. However, as in an R+-tree, object *D* is shared by both leaf nodes and both locks only affect their own granules. In this case, the GL/R-tree protocol does not provide sufficient phantom update protection for object *D*. One possible solution to this problem would be to lock objects rather than leaf granules. In this way the MBRs for the objects can be viewed as leaf granules and the external granules would be defined similarly for leaf nodes. Although this solution solves the above problem for the deletion (and updates), the object level locking substantially increases the number of locks. For example, if a search window were to return 10,000 objects, it would require 10,000 object level locks to be placed at the time of searching and released at the time of commit. Using coarse leaf granules, as proposed in the GL/R-tree, and assuming 100 maximum entries per

node and an average fill factor of 0.5, only 200 such locks would need to be requested and released. Therefore, for applications where selection is the predominant operation, locking at the object level may not be the desirable solution, and a new locking protocol is therefore required for multidimensional indexing trees such as R+-tree. For the ZR+-tree proposed in this research, the protocol also needs to deal with clipping of the inserted object, so a new concurrency control approach must be designed to provide phantom protection efficiently for indexing trees with object clipping.

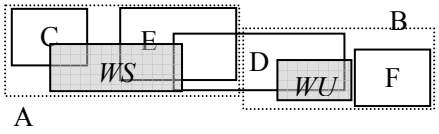


Figure 8. Example R+-tree for GL/R-tree Protocol.

Due to the complex structures that the techniques to enhance the R-tree's ability for moving objects bring, specific concerns are required to provide serializable isolation on these improved access methods. The existing concurrency control protocols for the R-tree are neither sufficient to protect the continuous queries, nor suitable to handle the R-trees with lazy update buffers. The former requires protection on multiple independent indices, whereas the latter needs to assure the consistency on both in-memory and on-disk indices. It is not a trivial task to fuse the techniques in the above areas into a real-world moving object management system that supports efficient continuous query. One focus of this work is to design concurrency control protocols for the R-tree-based access methods that support both frequent update and continuous moving query on moving objects.

2.2 Linear Spatial Index Related Techniques

In this section, the data access methods based on SFC and B+-trees are reviewed, and the concurrency control protocols designed for one-dimensional indexing structures are discussed. In addition, existing query processing approaches for moving objects are introduced.

2.2.1 Indexing Structure

Space-filling curves, as a linear mapping schema, have been extensively studied, e.g., the Peano-Z curve [50] maps data from the unit interval to the unit square; the Gray code curve [51] improves Z code by hashing; Hilbert [52] generalizes the idea to a mapping of the whole space. A historical survey and the major types of space-filling curves are introduced in [3]. Among the different types of SFCs, the Hilbert

curve has been shown to preserve the best data locality under most circumstances [53]. Several major space-filling curves, Hilbert, Peano-Z, Gray, Scan, and Sweep, have been systematically compared [20, 21]. Curve features in terms of the percentages of different segments have been analyzed, and five segment types (Jump, Contiguity, Reverse, Forward, and Still) have been defined to describe how the curves traverse the data space [21]. Normally, high percentage of Contiguity and Still is thought to be important to preserve locality. The results of the analysis show that the Hilbert curve has the lowest percentage of Jump and the highest percentage of Still in spaces with different numbers of dimensions, and the Peano-Z curve generally has the highest percentage of Contiguity. Furthermore, the Hilbert curve exhibits the best fairness in different dimensions of a space. On the other hand, the clustering properties of space-filling curves have been analyzed in [20]. The number of clusters in a multidimensional query range indicates the number of queries on one-dimensional space. In another word, the less clusters, the better locality of the spatial objects in that range is preserved in one-dimensional space. Both asymptotic analysis and exact analysis have shown that the Hilbert curve contains fewer clusters than the Peano-Z and Gray curves in both two-dimensional space and three-dimensional space [20].

Spatial operations have been studied on SFCs. Spatial join and range queries on Z-order have been proposed in [24]. The Gray curve has been claimed to outperform the Peano-Z curve on range queries in [26]. The Hilbert curve has been applied on range queries using a mapping tree [25]. Recently, an approach that calculates the Hilbert values via transformation between the Peano-Z curve and the Hilbert curve has been used to find nearest neighbors on the Hilbert curves [23]. Most of those approaches focus on efficiently determining the corresponding space-filling curve values for the cells in the query range. However, the processing part, i.e., efficient traversal of the B+-tree given curve values, still has space for optimizations, compared to the one-dimensional operations. There could be potential performance gain from optimizations in organizing the SFC-mapped one-dimensional queries before traversing the B+-tree. Furthermore, in some cases the original spatial queries designed for R-trees are not suitable to be directly applied to an SFC-based index. For example, NN search approaches on the R-tree [6, 7], and on partition-based data access methods [54] could not process k-nearest neighbor queries efficiently on SFCs, because the linear mapping provided by the SFCs cannot retain the entire spatial locality, even with the Hilbert curves. Furthermore, the general multi-step NN search [55] is not suitable for an SFC-based index, because it needs to find the number of objects in a certain set of leaf nodes on the B+-tree, and based on SFC, the leaf nodes only contain the cells and the links to the data pages.

2.2.2 Concurrency Control

Many concurrency control approaches [27-29, 56-58] have been proposed to support general concurrent search and update on B-trees. These approaches can be categorized as link-based and lock-coupling. The link-based approaches [27-29] have only one type of lock in all the operations, and their read operation does not require any locks. In these approaches, a lock-free read operation follows the links from root to leaf to find the correct nodes level by level, and traverses rightward if the current node does not contain the queried key because of reorganization. A complete set of concurrent operations on the B^{link}-tree, including read, insert, delete, modify, and reorganize, have been described in [27]. In this approach, the update operations (insert, delete and modify) only place locks on the leaf nodes they need to access, but not on any internal nodes. The modifier, which reorganizes the tree periodically, accesses the metadata of the index to merge and split the internal nodes correspondingly. In this way, this approach utilizes the left-to-right links between the nodes on the same level to implement a simple and efficient concurrency control protocol on the semi-dynamic B^{link}-tree. To transform the semi-dynamic approach to be dynamic, a symmetric concurrency control, which includes a two-phase merge algorithm to avoid periodically reorganize, has been proposed in [28]. However, in a multidimensional environment, the link-based concurrency control is not sufficient to assure the multiple B-tree searches contained in one spatial query occur at one time as required. In other words, it cannot guarantee the freshness of the search results from interfering update operations. The other category, lock-coupling approaches [56-58], applies at least two types of locks, read-lock and write-lock, to provide flexible concurrency control. These different locks are combined to make sure the B-trees are correctly and dynamically restructured, and help to perform recovery for aborted transactions. The R-tree family usually applies the lock-coupling approaches, e.g., the concurrency approach proposed for NN query on the R-tree [59], because it is difficult to define a global order for spatial objects. Lock-coupling approaches can provide more effective protection than the link-based protocols, but require complex maintenance to the locks on different levels of the indexing tree. A comparison between these two types of concurrency control approaches is illustrated in Table 2.

Table 2. Link-based Methods vs. Lock-coupling Methods.

Approach Type	**Lock types**	**# of locks**	**Overhead**	**Flexibility**	**Implementation**
Link-based	1	Low	Low	Low	Simple
Lock-coupling	≥ 2	High	High	High	Complex

2.2.3 Querying Moving Objects

Several spatial-temporal indexing structures [4, 30, 60, 61] based on B-trees have been proposed to manage moving objects and process spatial-temporal queries. Among these approaches, the B^x-tree [4] 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 on location and velocity, the B^x-tree not only can handle the queries on current locations, but also can answer the spatial queries for the near future. the BB^x-tree [30] 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 [61] improves the query performance of the B^x-tree on moving objects by indexing both the locations and velocities of objects. 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 [31] manages objects and queries by using hashing techniques, and incrementally processes positive and negative updates. Another approach, MAI [14], 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 [62] has been proposed to optimize the communication and query reevaluation costs due to frequent location updates.

This research on linear spatial indices focuses on providing serializable isolation for concurrent operations on spatial objects, especially on moving objects, by combining the B^{link}-tree and lock-coupling approaches.

Chapter 3. ZR+-TREE AND GLIP

This chapter provides an efficient solution for concurrent access methods on the R-trees with clipping technique for stationary objects. To support efficient concurrent access on stationary spatial objects, we proposes a Zero-overlap R+-tree, ZR+-tree, which compensates for this limitation of the original R+-tree in specific scenarios by eliminating overlaps of the leaf nodes. Furthermore, we have designed the concurrency control for the ZR+-tree, Granular Locking for clIPping indexing (GLIP), to provide phantom update protection, which also can be used in the R+-tree. The ZR+-tree, together with the proposed concurrency control, makes up an efficient and sound data access model for multidimensional databases.

3.1 Definition of ZR+-tree and Concurrency Control Protocol

Before proceeding to the details of the proposed data access framework, we define the notation that will be used throughout the rest of this document.

3.1.1 Terms and Notations

The terms used to describe the ZR+-tree structure are listed in Table 3. Suppose *T* denotes a ZR+-tree, then *T.root* refers to the root node of this tree. For each node *P* in *T*, *P.isLeaf* indicates whether the ZR+-tree node *P* is a leaf node or not, *P.level* gives the level of *P* in *T*, *P.entries* gives the current number of entries in the node, and *P.capacity* is the maximum number of entries the node *P* can hold. Note that by convention for all nodes *P* in *T*, *P.capacity* is the same. *P.mbr* gives the MBR for the node *P*, and is defined as an empty rectangle where *P* is NIL. For internal nodes, $P.child_i$ is an entry pointing to a ZR+-tree node which is *P*'s *i*-th child, and $P.rect_i$ gives the MBR for the *i*-th entry. For leaf nodes, $P.child_i$ gives the object pointed to by the *i*-th entry and $P.rect_i$ refers to the MBR for this entry. For each rectangle *R*, *R.l* gives the lower left corner and *R.h* gives the upper right corner.

As for the R+-tree, the ZR+-tree is height-balanced, so for each *P* in *T* where *P.isLeaf* is true, *P.level* is the same. Consequently, if *P* is an internal node, then for all $P.child_i$, $P.child_i.isLeaf$ is false or for all $P.child_i$, $P.child_i.isLeaf$ is true. As the data objects in ZR+-tree may be clipped, for leaf nodes, $P.rect_i$ may only indicate part of the MBR of the data object. Therefore, an object can be exclusively covered by multiple nodes. Furthermore, *P.mbr* must cover all the $P.rect_i$, no matter whether $P.child_i$ is an internal node or not.

Table 3. ZR+-Tree Node Attributes.

Term	Description
capacity	the maximum number of entries a node has
entries	the number of entries in the node
mbr	minimum bounding rectangle of the node, denoted by the bottom-left (l) and upper-right (h) vertices
level	the level of the node in the tree
child$_i$	the ith child of the node
rect$_i$	the MBR of the ith child of the node
isLeaf	true if the node is a leaf node

In the definition of R+-tree, for each internal node P, if P's children are not leaf nodes (all $P.child_i.isLeaf$ are false), then $P.rect_i$ encloses or covers $P.child_i.rect_j$ for all j. On the other hand, if P is the parent of some leaf nodes, then for all j, $P.child_i.rect_j$ need only overlap with $P.rect_i$. Note that the converse also holds. Therefore, if S is an internal node and $P.child_i = S$ (S is a child of P), then each $S.rect_i$ must be enclosed or covered by $P.rect_i$, and if S is a leaf node and $P.child_i = S$, then each $S.rect_i$ must overlap $P.rect_i$.

The presence of a standard lock manager [41] is presumed to support conditional and unconditional lock requests, instant duration, manual duration, and commit duration locks in GLIP. A conditional lock request means that the requester will not wait if the lock cannot be granted immediately. An unconditional lock request means that the requester is willing to wait until the lock becomes grantable. Instant duration locks merely test whether a lock is grantable, and no lock is actually placed. Manual duration locks can be explicitly released before the transaction is over. If they are not released explicitly, they are automatically released at the time of commit or rollback. Commit duration locks are released automatically when the transaction ends. As per the conventions in [9], five types of locks, *S (shared lock)*, *X (exclusive lock)*, *IX (Intention to set X locks)*, *IS (Intention to set S locks)* and *SIX (Union of S and IX lock)* are usually used in concurrency control protocols. In the proposed protocol, only *S* and *X* locks are used to support concurrent operations with a relatively simple maintenance process.

Also, the lock manager in GLIP is presumed to support the acquisition of multiple locks as an atomic operation. If this is not the case, such a procedure can be easily implemented by acquiring the first lock in a list unconditionally and all subsequent locks conditionally, and the procedure releases all the acquired locks and restarts if any of the conditional locks cannot be acquired. Furthermore, a transaction may place any number of locks on the same granule as long as they are compatible. The lock manager will place separate locks for each, and each lock will be a distinct lock even if the lock modes are the same. While releasing manual duration locks, both the lock granule and lock mode need to be specified.

3.1.2 ZR+-tree

R+-trees may be viewed as an extension of the K-D-B-trees [63] to cover rectangles in addition to points. In the original paper [11], the R+-tree was defined as having the following properties and characteristics:

1. A leaf node has one or more entries of the form (*oid, RECT*) where *oid* is an object identifier and *RECT* is the MBR (Minimum Bounding Rectangle) of the data object.
2. An internal node has one or more entries of the form (*p, RECT*) where *p* points to an R+-tree leaf or internal node *R*, such that if *R* is an internal node, then *RECT* is the MBR of all the (p_i, $RECT_i$) in *R*. On the other hand, if *R* is a leaf node, for each (oid_i, $RECT_i$) in *R*, $RECT_i$ need not to be completely enclosed by *RECT*. Each $RECT_i$ just needs to overlap with *RECT*.
3. For any two entries (p_1, $RECT_1$) and (p_2, $RECT_2$) in an internal node *R*, the overlap between $RECT_1$ and $RECT_2$ is zero.
4. The root has at least two children unless it is a leaf.
5. All leaves are at the same level.

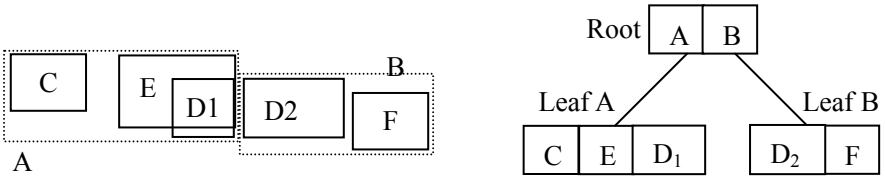


Figure 9. Example of ZR+-tree for the Data in Figure 6.

Some modifications must be made to the original R+-tree to make it suitable for the situations mentioned in Section 2.1.1. The proposed tree structure eliminates overlaps even among entries in different leaf nodes. It has therefore been named the Zero-overlap R+-tree (ZR+-tree). The essential idea behind the ZR+-tree is to logically clip the data objects to fit them into the exclusive leaf nodes. As this ensures zero overlap in the entire search tree, the structure and the operations become more orthogonal. Furthermore, this zero-overlap design can avoid the limitations associated with duplicating the links between objects that were discussed in Section 2.1.1. An example of the ZR+-tree that can be compared to the R-tree and R+-tree in Figure 6 is given in Figure 9, where object D is clipped to D1 and D2 to achieve zero overlap and avoid the construction limitations of the R+-tree.

The definition of the ZR+-tree is given in the form of a revised version to the earlier definition of R+-tree

by modifying properties 1 and 2 as follows:

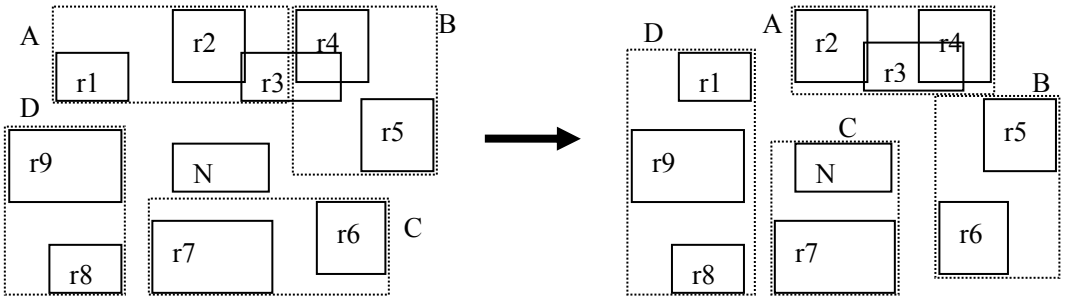
1. A leaf node has one or more entries of the form (*objectlist, RECT*) where *objectlist* gives the identifiers for each object that completely encloses or covers *RECT*. This approach is implemented in non-unique B-tree indices. Note that a single bounding rectangle with more than object *IDs* is still counted as a single entry, even though it requires extra space in the node. An alternative is to use a pointer as *objectlist* to an entry of a table that stores the corresponding object *IDs*.
2. An internal node has one or more entries of the form (*p, RECT*) where *p* points to an R+-tree leaf or internal node *R* such that *RECT* is the MBR of all the (p_i, $RECT_i$) in *R*. Thus, the definition of the R+-tree is more orthogonal by eliminating the difference in rules for the MBRs of leaf nodes and internal nodes. However, the MBR of an object may be fragmented such that the union of all the fragments equals the MBR of the object and each of the fragments may be inserted into the same or different leaf nodes.

Additional to the structure evolution, two operation strategies are proposed to improve the insertions on the ZR+-tree, as well as to refine the indexing tree.

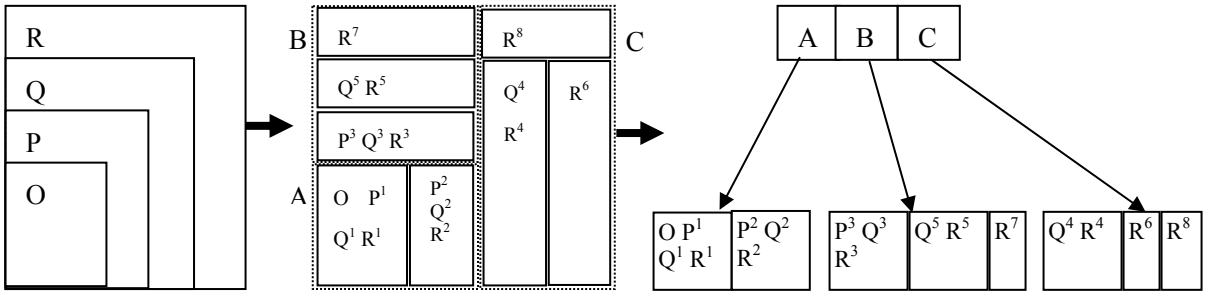
1. While performing an insert operation or a split operation, different plans are evaluated based on the number of object clipping occurrences, and the overall coverage. For an insert operation, each possible way to expand existing nodes to cover the new object is treated as a plan. Plans for split are the possible hyper-planes corresponding to any dimension to divide the node into two parts. The plan with the least object clipping, then with the smallest overall coverage, is picked for performing that operation.
2. Once a failure of insertion (as shown in Figure 10(a)) or a split propagation caused by updating has occurred, a clustering-based re-insert operation will be performed to optimize the distribution of the nodes. The re-insert will group the entries which are spatially nearby, and then form up new entries. The number of new entries should be the same as the number of old entries, or the number of old entries plus one. If the re-insert operation fails to enable the insertion of the proper object eventually, a compelled split, which requires object clipping, will be performed to accomplish the insert operation.

Figure 10(a) and (b) show the ZR+-trees corresponding to the R+-trees in Figure 7(a) and (b) that result from the above modification of properties. Note in Figure 10(a), a re-insert has been performed in order to build new entries. The 10 objects are clustered into 4 groups based on their positions. This new clustering of the entries avoids the deadlock situation. Although this strategy is not guaranteed to solve such a deadlock, it is able to alleviate the problem in most cases. In Figure 10 (b), if *P* is inserted after *O*, *P* will

need to be fragmented into three rectangles (*P1*, *P2*, *P3*) before it can be inserted. If after that *Q* is to be inserted after *P*, similarly *Q* will be fragmented into five rectangles, in which *Q1*, *Q2* and *Q3* are cut to correspond with *P*'s existing rectangles, while *Q4* and *Q5* are fragmented due to the rules of the rectangle regions. Similarly, *R* will be fragmented into 7 rectangles. In this way, the original entry of *O* is now holding the fragments of *P*, *Q* and *R*, and the whole node is easy to split because of the shape of the fragments.



(a) Clustering-based Re-insert in ZR+-tree



(b) Object Clipping in ZR+-tree

Figure 10. ZR+-tree Solution to the Problem in Figure 7.

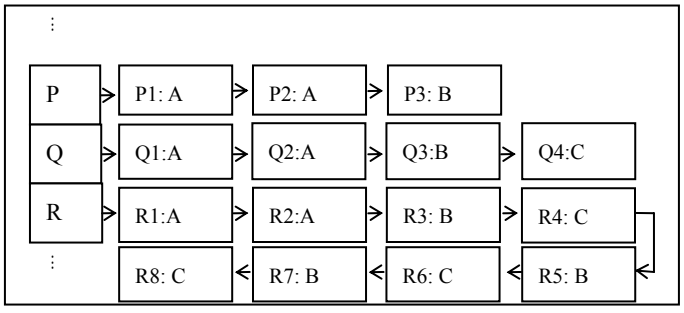


Figure 11. Clip Array to Link Objects in Figure 10 (b).

In order to support the proposed index tree, additional metadata is required to store the information concerning object clipping. While updating a data object, the operations need to know how many pieces it

has been clipped into and which leaf nodes they are located in, and then expand the operation to the remaining parts if necessary. An array of linked structures is designed to maintain the object information to enable such operations on a ZR+-tree. Each MBR of the clipped objects is added as an element of the array, and all pieces of the object entries will be linked one after another in this array element. The *IDs* of the leaf nodes that overlap the object also are stored along with the link, as shown in Figure 11. As only one MBR and several *IDs* for each clipped object are stored in this clip array, it is feasible to store the whole array in physical memory. Based on our experiments with real data, on average, each object is clipped to less than 1.5 segments, so it is reasonable to assume that each clipped object can use two double integers to denote the MBR and 16 integers as 8 links (2 *IDs* for each link). In this case, 100,000 objects occupy only 4MB, which is almost trivial compared to the memory size available in mainstream computers.

Note that a similar search algorithm to that used for the original R+-tree can be applied to the ZR+-tree, although new algorithms for insertion and deletion must be designed for the clipped objects. The algorithms for all these operations, along with those associated concurrency control, will be presented in Section 3.2.

3.1.3 Lockable Granules

Each leaf node in the ZR+-tree is defined as a lockable granule. We also define an external lockable granule for each ZR+-tree node as the difference between the MBR of the node and the union of the MBRs of its children. In order to reduce the overhead associated with lock maintenance, objects are not individually lockable. The clip array introduced as an auxiliary structure to store the clipping information for the objects in the ZR+-tree does not need to be locked, because the lock strategy on leaf nodes ensures the serializability of access on the same object, and updating on one object will not affect the other objects if the array is implemented as a link structure. Thus, in the case of the indexing tree in Figure 9, the leaf nodes *A* and *B*, *ext(A)*, *ext(B)*, and *ext(root)* are defined as lockable granules. *ext(A)* covers the region $A.mbr - (C.mbr \cup D_1.mbr)$, and *ext(root)* covers the region $MBR(A.mbr \cup B.mbr) - (A.mbr \cup B.mbr)$. The above lockable granules cover the entire space covered by the MBR of the root. However, all of these lockable granules do not fully cover any search windows that are partially or fully located outside the MBR of the root. One option is to define *ext(T)* as a lockable granule that covers all such space. Another option is to define the *ext(root)* itself to include *ext(T)*. For inserting objects into such space, either approach leads to the same level of concurrency, since any insertion outside the root's MBR leads to the growth of the MBR for the root and thus conflicts with *ext(root)*. However, for select and delete operations, *ext(root)* and *ext(T)* need not conflict. For example, a delete operation that overlaps with the lock granules *C*, *ext(A)* and *ext(root)* can coexist with a select operation that overlaps with *E* and *ext(T)*. Thus, defining *ext(T)* as a

separate lockable granule leads to greater concurrency. It also effectively handles situations where the tree is empty and the root is NIL. Summarizing the above analysis, the lockable granules in ZR+-tree for GLIP are defined as all the leaf nodes, external of the nodes, and external of the tree.

3.2 Operations with GLIP on ZR+-tree

To support ZR+-tree, a granular locking-based concurrency control approach, GLIP, that considers the handling of clipped rectangles, has been designed to meet the following requirements:

1. Algorithms for the ZR+-tree operations.

Select for a given search window: This is presumed to be the most frequent operation. This operation could result in the selection of a large number of objects, though this may be only a fraction of the total number of objects. Hence, it is desirable to have as few locks as possible that must be requested and released for this operation.

Insert a given key: Having redefined the properties of the R+-tree with clipped objects, a new algorithm must be provided for insertion in a ZR+-tree.

Delete objects within a search window: Since an object in the ZR+-tree may be clipped and the search window might not select all the fragments of a given object, the algorithm is required to delete all fragments of the selected objects in order to maintain data consistency.

2. The locking protocol should ensure serializable isolation for transactions, so that it will allow any combination of the operations described above to be performed concurrently.

3. The locking protocol proposed above also should ensure consistency of the ZR+-tree under structure modifications. When the ZR+-trees nodes are merged or split in cases of underflow or overflow, the occasionally inconsistent state should not lead to incorrect results.

4. The proposed locking protocol should not lead to additional deadlocks. For instance, if transaction *T1* deletes *A* and reads *B*, and transaction *T2* deletes *B* and reads *A*, and both the deletions happen before the two reads, a deadlock is acceptable since it is caused by the transactions, rather than the locking protocol.

Details of the algorithms are provided in the following subsections with formal algorithm descriptions.

3.2.1 Select

The select operation, shown in Algorithm 1, returns all object *IDs* given a search window W. It is necessary to place locks on all granules that overlap with the search window in order to prevent writers from inserting or deleting into these granules until the transaction is completed.

Algorithm Select(W, T)

Input: search window *W*, ZR+-tree *T*
Output: set of object *ID O*

```
O := {}; P := T.root
If (P is NIL) or (not(P.mbr ∩ W))
   return O
End If
If W ∩ P.mbr<>P.mbr    //Root does not cover W
   Lock(ext(T), S, Commit) //Lock external of tree
End If
Lock(ext(P), S, Manual) //Lock the root
Stack L := {( P.mbr ∩ W, P)}
 //Traverse the indexing tree and lock/unlock the visited nodes
Loop until L is ∅
   (R, P) := L.pop
   For each i in P.rect_i
      If P.rect_i ∩ R Then
         If P.isLeaf Then
            O := O ∪ P.child_i    //Add the objects that are not yet in results
            Unlock(P, S)
         Else
            If P.child_i.isLeaf Then
               Lock(P.child_i, S, Commit)
            Else
               Lock(ext(P.child_i), S, Manual)
            End If
            L.push({(P.rect_i ∩ R, P.child_i)})    //Put the child of P in stack
         End If
         R := R - P.rect_i
      End If
   End For
   If (not P.isLeaf) and (R = ∅)
      Unlock(ext(P), S) //Release S Lock on ext(P) to S Lock if it does not overlap R
   End If
End Loop
Return O
```

Algorithm 1. Search Algorithm.

Selection starts by checking whether the search window overlaps with *ext(T)*. If it does, it places a shared

lock on *ext(T)*, thus preventing a writer from inserting data into this space. It then performs a breadth-first traversal starting from the root node and traversing each node whose MBR overlaps with the search window. For an internal node that overlaps with *W*, *S* lock is placed on its external area. This lock is released when all of its child nodes and its external granule have been inspected and locked if necessary. For each internal node, if the MBRs of its children do not fully cover the search window *W*, an *S* lock will be kept on the external granule for the node in order to prevent writers from modifying the region. This ensures consistency of the tree, as it prevents writers from modifying the internal node until all the child nodes have been inspected and protected properly. As discussed earlier, in order to reduce the number of locks that must be placed and released, we neither perform object level locking, nor lock the corresponding objects in the clip array for the select operation. Instead, shared locks are placed on the leaf nodes that overlap with *W*. Since the same object *ID* may recur in the same leaf node or across different leaf nodes, a set of object *IDs* is maintained to avoid returning the same object *ID* more than once. This is consistent with the expected result from a select statement, and reduces the burden on the caller. Finally, all the *S* locks on the granules that overlap with *W* are released once the search is done.

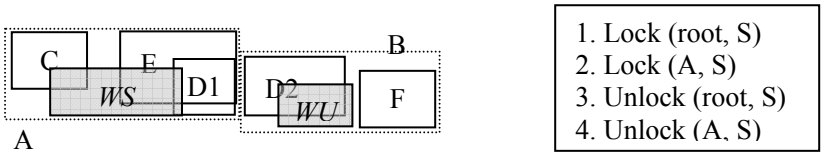


Figure 12. Example of Locking Sequence for *WS*.

Figure 12 illustrates the lock management for the window query in the figure. For a search window *WS* that overlaps with *C*, *E* and *D*, initially an *S* lock will be placed on the root. Then an *S* lock will be placed on leaf node *A* and the lock on the root will be released. This prevents any other transaction from modifying the root (by placing an *X* lock on it) until all its children have been inspected. After the lock on root has been released, the entry for node *B* in the root can be modified (as long as the modification does not result in overlap with *A*). Thus, manual duration *S* locks are used to maintain consistency while at the same time maximizing the degree of concurrency.

3.2.2 Insert

As compared with R+-trees, the insert operation for ZR+-trees (Algorithm 2) has more considerations. To illustrate the insert operation, let us name the MBR of the object to be inserted as *W*. First consider all the fragments of *W* that do not overlap with any other object MBRs. These fragments need to be inserted into the leaf nodes of the tree. On the other hand, the fragments that intersect with existing object MBRs may

result in clipping the existing MBRs if they are not equal. Consider the object MBRs in Figure 10 (b). If *P* is inserted after *O*, *P* will need to be fragmented into three rectangles (P_1, P_2, P_3) before it can be inserted. Similarly, if *Q* were to be inserted after *P*, the same clipping still would be required. The number of fragments that an insertion will cause is a function of the gaps in the object MBRs.

The insert operation without concurrency control protocol proceeds as follows. First, a breadth-first traversal is performed from the root of the tree. When *W* is found to be covered by node *N* but not any single child of *N*, the child nodes of *N* are selected to extend if *N* is an internal node. If *S* is the set of child nodes for *N*, *S* is partitioned into two sets, *S1* and *S2*, such that *S1* contains all the child nodes whose MBRs need not be changed, and *S2* is the set of nodes that need to be changed to cover *W*. In order to select the appropriate set of nodes to extend the MBRs, it is the solution with the fewest nodes involved and then the smallest coverage that is adopted as the heuristic. This leads to a relatively small tree and small coverage area which help achieve good search performance by fitting more index into memory and eliminating paths as quickly as possible.

No granules are locked during this traversal, although all the granules that overlap with *W* are remembered. After the traversal, *X* locks are placed on all of these lock granules in an atomic fashion. If the locks are successfully acquired, the actual insertion then can be performed. Since the *X* locks are retained on all these granules until the transaction is complete, this guarantees that any other operations that need to traverse any part of the path impacted by the insertion will need to wait till the transaction is committed. At the same time, since any active selection will hold *S* locks on all the granules that it has covered, and update operations always attempt to place *S* or *X* locks on the area they intersect, an insert operation only will be performed when no active insertions, deletions or selections that overlap with the insertion are present, thus guaranteeing the serializability.

There is still a risk that insertion transactions *T1* and *T2* perform at the same time, following intersecting paths and then waiting for *X* locks. If no selections are active and *T1* acquires the *X* locks first, it will perform its insertion and then commit. Now *T2* can acquire the *X* locks but the path it had previously traversed is *dirty*. In order to prevent *T2* from performing insertion on this dirty data, a version number is maintained for each node. All *X* lock requests implicitly pass the current version of the node for which the *X* lock is requested. When the *X* lock becomes grantable, the current version number is compared with the version number at the time of the request. If they do not match, the lock is released and a dirty signal is returned, causing the insert procedure to be restarted. While this is expensive, it is common even in commercial systems where version-based or timestamp-based concurrency control protocols are used. For

example, in [33] such a restart is performed even for select operations. However, we prefer not to require the selection to restart, as our protocol generally favors readers.

Algorithm Insert(W, O, T)

```
Input: key W, object O, ZR+-tree T, queue of X locks to request M
Output: NIL

L := {}; P := T.root; M := {}; S2 := {}
// Record required locks
If W ∩ P.mbr<>P.mbr    //root does not cover W
   M.enqueue({ext(T), X, Commit})
End If
L.enqueue({P, W})
Loop until L is ∅
      (P, R) := L.dequeue
      If P.isLeaf
         S2 := S2 + {P, R}
      M.enqueue({P, X, Commit})
   Else
      If P.mbr covers R and !(∃ i, P.rect_i covers R)
         M.enqueue({ext(P), X, Commit})
         S := minExtend(W, P) //Choose node list S in P to extend to include W with minimum cost
                              and update their MBRs (See Algorithm 3)
         L.enqueue({each node in S and its extended MBR})
         break
      Else
         n := P.child_i | P.child_i covers R
         L.enqueue(n, R)    //Traverse down
      End If
   End If
End Loop
// Request locks and insert object, or re-do if conflict occurs
If LockAll(M) //Request all the X locks and check version
  For every pair (P, R) in S2
      P.child(P.entries) := O
      P.rect(P.entries++) := R
      If R<>W    //The object is clipped
      StoreClipArray(O, R, P) //Store object in clip array
      End If
      If P.entries > P.capacity    //Overflow
      Split(P) //If splits bubble up to a node not in M then add the node to M and restart from LockAll
      End If
  End For
Else
   Insert(W, O, T)    //Restart insert
End If
Return
```

Algorithm 2. Insert Algorithm.

```
Function: minExtend(W, T)
//Chose a set of nodes to include object W with minimum cost
Input: key W, ZR+-tree T
Output: NodeList S

N := {}; SS := {}
Loop until N = {T.child_i}
   P := the nearest child of T to W and not in N
   N := N+P
   For all combinations of nodes in N
      If the set of nodes SN can extend to cover W without overlapping each
other
         SS := SS+SN;
      End If
   End For
   If SS <> ∅
      S := solution in SS with least nodes and then least coverage
      Break
   End If
End Loop
If SS = ∅
  Lock(ext(T.root), X, Manual) //Lock the branch for structure refinement
  T' := Re-insert(W, T)
  Unlock(ext(T.root), X)
  S := minExtend(W, T')
End If
Return S
```

Algorithm 3. minExtend Function.

Conflicts between insertions that could cause deadlocks are avoided by simultaneously requesting all the *X* locks needed by an insertion. With the proposed protocol, as part of the insert operation, the insertion only holds *X* locks once and requires no locks before or after that. Thus, no deadlocks can be induced using this protocol, since for any deadlock to occur, the protocol would need to request a conflicting lock while holding other locks. If the *X* locks are not requested at the same time, and the insertion were to place *X* locks on each lockable granule it traverses, chances are, an *X* lock has been propagated bottom-up by a node split in an insert operation, and meanwhile a select operation attempts to acquire an S lock on the same node. This would cause a deadlock. One alternative is to place *IS* locks for selections and *SIX* locks for insertions, which would prevent the above sequence from causing a deadlock. However, this solution is not optimum for the following reason. If the insertions were to keep on holding *SIX* locks on higher-level granules, then no other insertion could be carried out at the same time, thus resulting in a low degree of concurrency. To avoid this, the insertion could release the *SIX* locks on the higher-level granules if the traversal indicates that they need not be modified. However, a split operation at a lower level may cause a

parent node to be updated. Thus, the insertion would never be able to guarantee that an internal node is not to be updated. Even if the possibility of splits while releasing the *SIX* locks is ignored, and a split indeed propagates along the tree, a protocol would still be required to handle the case that the parent node may be dirty. Thus, in order to simplify the protocol, a lock-at-one-time approach is adopted for all cases.

To conclude the insert algorithm as shown in Algorithm 2, the actual insertion is performed as follows. Pending insertions into all the leaf nodes are performed first. At this point, nodes that overflow are not split but only marked for splits. Via the minExtend function (shown in Algorithm 3), the nodes in *S2* then are expanded to include the new object *W* following an optimal plan with fewest nodes and smallest area involved. The node expansion is only logical, since they have not yet been locked. Once the expansion fails, a re-insert function (to be introduced in the next subsection) will be invoked to reconstruct *S2*. After the expansion, the new object *W* is segmented into pieces that can be covered by the *N* nodes, where *N* is the size of *S2*. This process repeats until the segments of *W* have been inserted into leaf nodes. The resulting leaf nodes may overflow after inserting *W*. Again the overflowing nodes are not physically split but only marked for splitting. Since *S2*.MBR does not overlap with the MBR of any of its siblings, splitting *S2* into nodes will only produce nodes whose MBRs will not overlap with their siblings as long as the split does not extend the MBR of the splitting region. The split algorithm presented in [11] guarantees this. The node insertion is now completed and all the nodes that have been remembered for lock will be *X* locked. The protocol then splits each leaf node marked for splitting, and inserts the new leaf node in the lowest level internal nodes. If this insertion causes an overflow in the lowest level internal nodes, again they are not split immediately but only marked for splitting. Once all the marked leaf nodes have been split, any lowest level internal nodes that may be marked for splitting are split, and this splitting may propagate the tree as required. Since not all the internal nodes are locked, this may cause the split to propagate to an internal node that has not been locked. In this case, this internal node is added to the list of nodes that require *X* locks, and the tree is restored to its state before the insertion. Then the process is repeated as it waits for locks on all the nodes, and the insert operation is performed as described above.

Clearly, if the select requests from other transactions continue to arrive while the insertion is waiting for the *X* locks, it is possible that the transaction that is waiting to insert never acquires its lock, resulting in starvation. To avoid this, a scheduling mechanism is used so that *S* locks are granted on resources which other transactions are waiting for an *X* lock on if, and only if, the transaction requesting the *S* lock arrived before the request for the *X* lock. The details of such a policy are not discussed in this research, however, interested readers may refer to [64].

3.2.3 Re-insert

In some cases, the basic insertion will fail (as shown in Figure 7(a)) because of the complex spatial relationships among existing nodes in *S2*. Moreover, propagated splits caused by updating are hard to avoid in database updating operations. A re-insert function is therefore necessary in order to solve or alleviate these problems. The idea of re-insert is to break up the existed nodes and form new nodes rationally, based on their spatial locations. Compared to the reinforced insert operation in R*-tree, this re-insert function focuses on redistributing multiple nodes, rather than optimizing the groups within one node. And eventually, the re-insert method guarantees the success of the insert operation by a compelled split.

Specifically, as shown in Algorithm 4, the re-insert function works as follows. Given a set of entries from *K* nodes, a clustering algorithm is used to generate the center entries of *K* clusters of the entries. These *K* center entries are used to form *K* nodes as the first level of a subtree. Consequently, the remaining entries are inserted into this subtree according to their distance from the center entries. After inserting all the entries, these nodes will be put back to replace the original nodes in the ZR+-tree. If after this stage, the insertion still fails, a compelled split is performed to split one of the *K* nodes, so that a subset of the *K* nodes can be extended to cover the new object. During the processing of re-insert, an *X* lock will be requested on the parent of the *K* nodes by its invoker, in order to protect this sub-tree from concurrent update operations.

```
Function: Re-insert(W, T)
//Re-arrange the entries in T according to their distribution
Input: key W, ZR+-tree T
Output: ZR+-tree T'

S := {child nodes of T.root}; C := Centroid(T.root.rmb); T':= ∅; SP:= ∅
P := the farthest mbr to C in S
SP := SP+P
For N:=2 to S.size
  P:= the farthest mbr to SP in S
  SP := SP+P
End For
T' := construct(SP) //Construct S.size children using each node in SP as the
                    only entry in one child
For every child n of T but not in SP
   Insert(n.mbr, n, T')
End For
T' = Compelled-split (W, T) //Split a certain child of T, to enable the insert of W
Return T'
```

Algorithm 4. Re-insert Function.

The classical clustering algorithm k-means is used for this clustering task because the number of clusters is fixed. Other clustering algorithms that can return a fixed number of clusters also may be applied in this function. One essential step to reduce the complexity of the clustering is to choose appropriate *K* seeds to start the *K* clusters. An optimal strategy is to select *K* seeds that are as far away from each other as possible. A similar idea for this step was applied in the CURE clustering algorithm [65].

This clustering-based re-insert function can group the entries according to their distributions. With the compelled split, this function can guarantee to avoid the failure of insertion, and can alleviate propagated splits. Furthermore, the tree structure will be refined after applying the re-insert function, because the affected objects are more likely to be grouped into their natural spatial clusters, regardless of the order of insertions.

3.2.4 Delete

The delete operation, as shown in Algorithm 5, works in a way similar to the insert operation. For the delete operation, since the same object may be fragmented and stored in multiple leaf nodes, it is necessary to ensure that all the fragments of an object are deleted. A delete window *W* may not select all fragments of the object, and thus deleting only the fragments that match the delete window can leave residual fragments. As addressed in Section 3.1.2, a clip array is maintained to store object *IDs* and pointers to the leaf nodes that store the fragments of the object. First, all object *IDs* that match the delete window are selected. The corresponding elements in the clip array then are read to find all the fragments in other leaf nodes, after which the object deletion is performed. However, it is wasteful to read the clip array for each selected object, because in many cases the object MBR may not be fragmented in the tree at all. An optimized strategy is to store a bit to indicate whether the MBR in the leaf node is the true object MBR or not. The algorithm thus needs to read the clip array only when the search window selects a fragmented MBR.

From the point of view of concurrency, all locks are requested using a single atomic operation. There are no locks requested before or after this lock request, thus avoiding deadlocks. Since any paths that have been traversed before the lock is requested may be dirty, similar to the insert operation, once a dirty signal is received from the lock operation, the delete process will restart. As it can be exactly determined whether the deletion will cause a node to underflow (i.e., when the number of entries is 0 after deletion) or not, this knowledge also can be used to add lock requests for the external granules of appropriate internal nodes.

Algorithm Delete(W, T)

```
Input: search window W, ZR+-tree T
Output: NIL
O := {}; O1 := {}; P := T.root; V := {}; M := Clip Array; Stack L := {( P.mbr ∩ W, P)}
If (P is NIL) or (not(P.mbr ∩ W))
   Return
End If
// Record required locks
If W ∩ P.mbr<>P.mbr    //Root does not cover W
   V.enqueue({ext(T), S, Commit}) //Lock external of tree
End If
Loop until L is ∅  //Traverse the indexing tree
   (R, P) := L.pop
   For each i in P.rect_i
      If P.rect_i ∩ R Then
         If P.isLeaf Then
            O := O ∪ P.child_i    //Add the objects that are not yet in results
            O1 := O1∪ {leaf nodes in M that covers P.child_i} //Add leaf nodes from the object link in clip array
         Else
            If P.child_i.isLeaf Then
               V.enqueue({P.child_i, X, Commit})
            End If
            L.push({(P.rect_i ∩ R, P.child_i)})    //Put the child of P in stack
         End If
         R := R - P.rect_i
      End If
   End For
   If (not P.isLeaf) and (R ≠ ∅)
      V.enqueue({ext(P), S, Commit}) //S Lock on ext(P) if it overlaps R
   End If
End Loop
For every node n in O1 //Lock all the leaf nodes that cover the objects to be deleted
   V.enqueue({n, X, Commit})
End For
For every internal node n in T whose MBR will shrink or be removed after deleting set O
   V.enqueue({ext(n), X, Commit})
End For
// Request locks and delete object, or re-do if conflict occurs
If LockAll(V) //Request all the locks and check version
   For each object n in O
      Delete n in the leaf nodes in O1; Delete n in M
   End For
   For each underflow leaf node n in O1
      Merge(n) //Propagated up if necessary
   End For
Else
   Delete(W, T) //Restart the delete
End If
Return
```

Algorithm 5. Delete Algorithm.

Since the delete operation requests *X* locks on the leaf nodes that contain the objects to be deleted, it will conflict with the *S* locks placed by the select operation. Once underflow occurs, *X* locks will be placed not only on the underflow node, but also on its parent, as long as its MBR needs to be shrunk or removed because of the underflow. Thus any search that commits after the commitment of the deletion will not retrieve the objects affected by this deletion. The delete operation also requests *S* locks on *ext(P),* where *P* is an internal node, and *ext(P)* overlaps with the delete window and is not exclusively locked. Therefore, no new objects that intersect with *ext(P)* can be inserted before the commitment of this deletion. While accessing the clip array to find fragments of the selected objects, *X* locks also will be requested on the leaf nodes that cover these fragments, thus this operation provides phantom access protection.

3.3 Analysis

ZR+-trees guarantee that if a query window is entirely contained in the MBR of a leaf node in the tree, only one search path is followed. And it also is assured that only one search path will be followed for point queries. Neither of these is always true in R-trees. Therefore, given an R-tree and a ZR+-tree with the same height, the ZR+-tree is likely to provide better search performance, similar to that of the R+-tree. Not only is following multiple paths likely to be wasteful, but a search in an R-tree also would result in a point query locking multiple leaf granules, thus reducing concurrency. The ZR+-tree offers better concurrency. Compared to the R+-tree, the ZR+-tree refines the node extension function in insertion, applies the re-insertion approach, and adopts the orthogonal object clipping technique. In this way, the ZR+-tree optimizes tree construction and removes the insertion and splitting limitations, which make it more stable than an R+-tree.

According to the definition, the number of entries in ZR+-trees may be larger than the number of actual objects due to fragmentation. These extra entries lead to additional space requirements for the ZR+-tree and also might increase the height of the ZR+-tree, which would degrade the efficiency of the search operation. In the worst case, if the total number of leaf nodes in the ZR+-tree that can be extended to cover part of the inserted object *W* without overlapping other nodes is *N*, neglecting potential splits, *W* will need to be fragmented into at most *N* fragments. Note that this worst case is applicable only when no fragments in *W* that are covered by extending a leaf node in *N* can be covered by extending another leaf node in *N*. When fewer leaf nodes are covered by the inserted window, the amount of fragmentation due to the insertion decreases. Furthermore, if the corresponding segments from different objects have exactly the same MBR, they are treated as a single entry in a leaf node. This approach keeps the number of entries in the ZR+-tree similar to or even smaller than the R+-tree, which has been verified by our experiments. As a result, a

ZR+-tree where the size of the dataset varies exponentially could be expected to increase the height linearly or even less, given a large fanout. That matches the results of the experiments shown in the next section: R-trees, R+-trees and ZR+-trees usually have the same height for the same dataset, with suitable capacity and fill factors.

It is significantly more complex to implement insert and delete operations for ZR+-trees. These operations also consume extra CPU cycles and I/O operations when compared with R-trees and R+-trees. Thus, the insert and delete operations could be expected to be slower than their R-tree and R+-tree counterparts. However, the complexity of the algorithm implementation itself is negligible for practical applications, if the increase in performance for the select operation is significant, especially since the implementation is a one-time cost. To select the appropriate index structure, an application would estimate the execution probability for each operation. If the application is searching-predominant, the ZR+-tree would be a better choice than either the R-tree or the R+-tree.

When used with the proposed GLIP protocol, the ZR+-tree access method meets the requirements of serializable isolation, consistency, and no additional deadlock. Specifically, serializable isolation is guaranteed by the strategy of requesting *S* locks on reading and *X* locks at the same time on updating. The consistency requirement is ensured by implementing version checking and restarting the insertion or deletion when the version does not match (as described in Section 3.2.1). Finally, the strategy of lock-at-one-time for insertion and deletion makes sure that no additional deadlocks occur. Benefiting from the above design, phantom protection is provided by the ability to lock on different granules.

As the proposed GLIP protocol takes into account object clipping, it can be extensively applied in an R+-tree and its variants. If it is applied in an R+-tree, the necessary modification will be to simplify the clip array until it only contains references to the corresponding leaf nodes that cover the same object. Because an R+-tree uses the reference to a complete object as each entry in the leaf nodes, with this simple change, GLIP provides phantom protection in an R+-tree.

Summarizing the above analysis, the ZR+-tree and its concurrency control GLIP can be used in an efficient and stable multidimensional access method which enables concurrent operations and is expected to outperform existing methods for searching-predominant applications.

3.4 Experiments

To evaluate the performance of the ZR+-tree and the corresponding concurrency control mechanisms, two sets of experiments were conducted as illustrated in Figure 13. The first set compared the construction and query performance of the ZR+-tree, R+-tree, and R-tree, while the other compared the throughput of concurrency control for the ZR+-tree and the R-tree. The design of experiments consisted of four components: selecting/generating benchmark datasets, constructing multidimensional indices, executing query operations, and measuring respective performance. The experiments made comparisons among ZR+-tree and various indexing trees using three benchmark datasets, namely major roads in Germany (28,014 rectangles), roads in Long Beach County, California (34,617 rectangles) (Figure 14), and a synthetic dataset (50,000 rectangles). In both of the real datasets [66], rectangles were used to indicate segments of the roads. Relatively speaking, the data distribution of the roads of Long Beach County was skewed, while the roads in Germany were more globally uniform in distribution. The synthetic dataset contained equal-sized rectangular objects that were uniformly distributed with a tunable density, which means that every point in the space was covered by a certain number of rectangles. In the experiments, we set this number as 3. In other words, any point in the data space is covered by three equal-sized rectangular objects. As shown in Figure 13, indexing trees were built for these datasets with varying size and controllable capacity and fill factors. In the query operation stage, some data was randomly taken from each of the above datasets for insertion during the experiments. The queries to be executed in both sets of experiments were generated by randomly choosing the query anchor from the data file and generating a bounding box with varying query window size. The numbers of disk accesses in execution were collected as the measure in the first set of experiments. In the second set of experiments, the write probability and concurrency level were changed in order to obtain the throughput.

The experiments were conducted on a Pentium 4 desktop with 512MB memory, running a Java2 platform under Windows XP. The implementations of the R-tree, R+-tree, and ZR+-tree were all based on the Java source package of the R-tree obtained from R-tree portal [5].

The first set of experiments evaluated the construction and query performance of the ZR+-tree. In these experiments, different data sizes were selected to construct the ZR+-trees, R-trees, and R+-trees. The disk accesses of the point queries were recorded by varying the number of rectangles. Additionally, the standard deviations of the disk accesses were calculated to compare the stableness between the ZR+-tree and the R+-tree. Consequently, queries with different window sizes were executed on the constructed trees in order to record the execution time. From a comparison of the algorithms, both the point query and window query

performances of the ZR+-tree are expected to be better than those of the R-tree. The number of disk accesses in this set of experiments was computed to be the average value of 1000 random queries in order to reduce the impact of uneven distribution of the data.

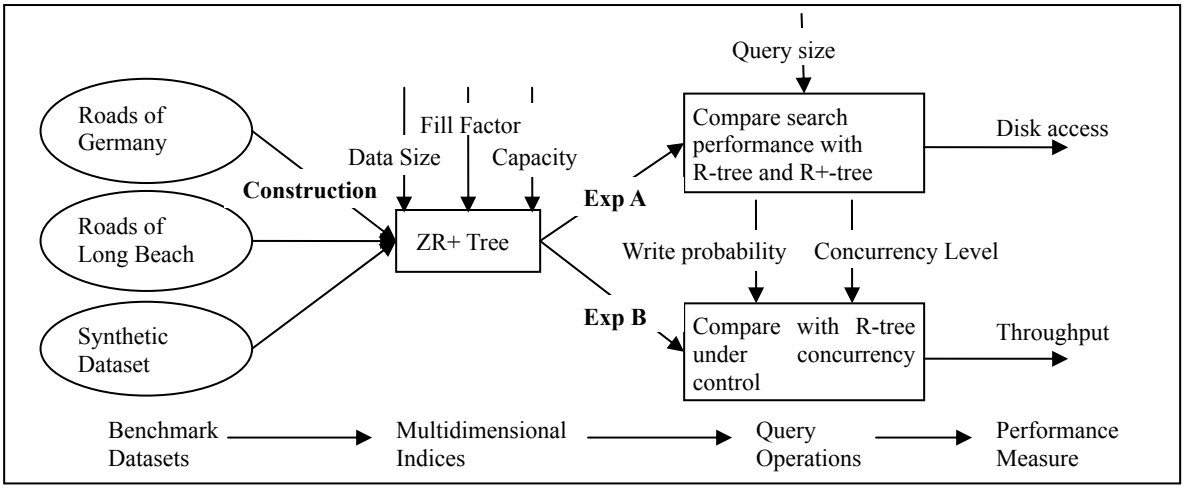
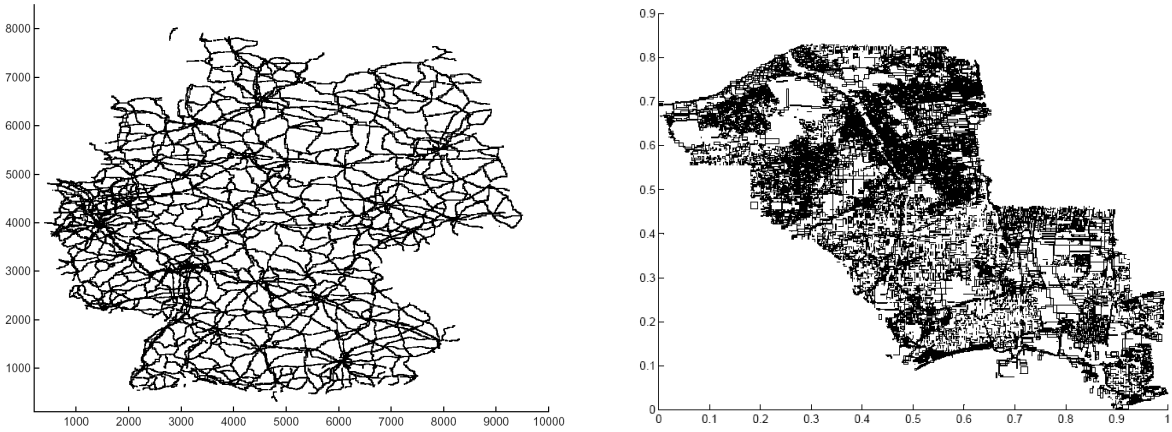


Figure 13. Experiments Design.



(a) Major Roads of Germany (b) Roads of Long Beach County

Figure 14. Datasets.

The second set of experiments evaluated the throughput of GLIP on the ZR+-tree by comparing it with dynamic granular locking on the R-tree [9]. The throughput of the two trees was compared under different write probabilities and concurrency levels.

3.4.1 Query Performance

Table 4. Construction Costs of ZR+-tree, R+-tree and R-tree.

	Roads of Germany (28,014 rectangles)				Synthetic Dataset (50,000 rectangles)			
	# of nodes	**# of leaf nodes**	**Page access in construction**	**Time Cost**	**# of nodes**	**# of leaf nodes**	**Page access in construction**	**Time Cost**
ZR+-tree	667	654	(I)122744 (O)73597	4.1min	1449	1427	(I)298593 (O)179023	10.3min
R+-tree	689	677	(I)126119 (O)77390	5.1min	1517	1498	(I)307324 (O)187861	11.9min
R-tree	699	678	(I)92842 (O)51864	0.3min	1154	1119	(I)228426 (O)133879	0.72min

Point query and window query operations were executed on the R-tree, R+-tree and ZR+-tree in order to compare the query performance. In this set of experiments, following the conventions, the capacity of the index trees was set to 100, the fill factor was 70%, and the data size and query size varied. The density of the synthetic data was set to 4. Building the three types of indexing trees on two real datasets, the height of the trees was always three, although the R-tree had the least number of entries in leaf nodes. The comparison among tree constructions of ZR+-tree, R+-tree and the R-tree on two datasets is illustrated in Table 4. The numbers of leaf nodes, the numbers of total nodes, the numbers of page accesses for reading (I) and writing (O), and the construction time are listed in the table for comparison. As shown in the table, the number of entries in leaf nodes of ZR+-tree was less than R+-tree by around 5% in both datasets, and less than the R-tree in Roads of Germany with a similar difference. Furthermore, ZR+-tree cost about 4% less I/O than R+-tree in both datasets. Because of the sophisticated insert, split (possible extend and split solutions are considered on two directions of each dimension), and re-insert algorithms in ZR+-tree, it gained better construction performance than R+-tree. Every object in R+-tree was duplicated in 1.53 leaf nodes on average, while each object in ZR+-tree was clipped to 1.49 segments on average. However, on the construction aspect, the R-tree still cost about 1/3 less I/O than ZR+-tree. In terms of time consumption, ZR+-tree and R+-tree were about 15 times higher than the R-tree, which reflects the time-consuming updating process in object clipping structures. Although similar construction time was cost by ZR+-tree and R+-tree, ZR+-tree cost slightly less than R+-tree because of less I/O accesses. That means the increased complexity in ZR+-tree insertion can be totally covered by its optimized tree structure. The costs on query aspect are shown in the following subsections.

3.4.2 Point Query

According to the design, the performance of point queries on a ZR+-tree should be better than that on an R-tree and comparable to that on an R+-tree. Figure 16 compares the number of disk accesses of point queries for each of the three indexing trees, as well as the standard deviation of disk accesses for the ZR+-tree and the R+-tree. The left-hand-side figures show the number of average disk accesses on the y-axis, and the size of datasets on the x-axis. The right-hand-side figures plot the standard deviations on the y-axis and the size of datasets on the x-axis. While the disk accesses of the R-tree increased along with the size of the dataset, the point query performance of the ZR+-tree and the R+-tree remained much lower than that of the R-tree as the number of objects increased. In both the roads of Long Beach County and the synthetic dataset, the number of disk accesses of the ZR+-tree remained almost constant in our experiments, which indicates that its performance is quite scalable. Interestingly, while constructing R+-trees, the program encounters a construction failure when the data size reaches around 19000 because of an insertion deadlock in the roads of Long Beach County dataset. To make the comparison complete, the troublesome data was removed and the repaired R+-trees were used (as the curve in Figure 16(b)). Figure 15 shows the deadlock situation in detail, where the shaded rectangle indicates the object to insert, and the gray rectangles are the internal nodes in the R+-tree. In this situation, the nodes cannot be extended to cover the object without overlapping with each other. Although the I/O costs of the ZR+-tree and the R+-tree are similar, the ZR+-tree constantly achieves lower or equal (only twice) standard deviations in all three datasets, which indicates that the ZR+-tree processes point queries more stably than the R+-tree. An examination of these outputs shows that for most of the cases tested, the point query performance of ZR+-tree is much better than the R-tree and more stable than the R+-tree.

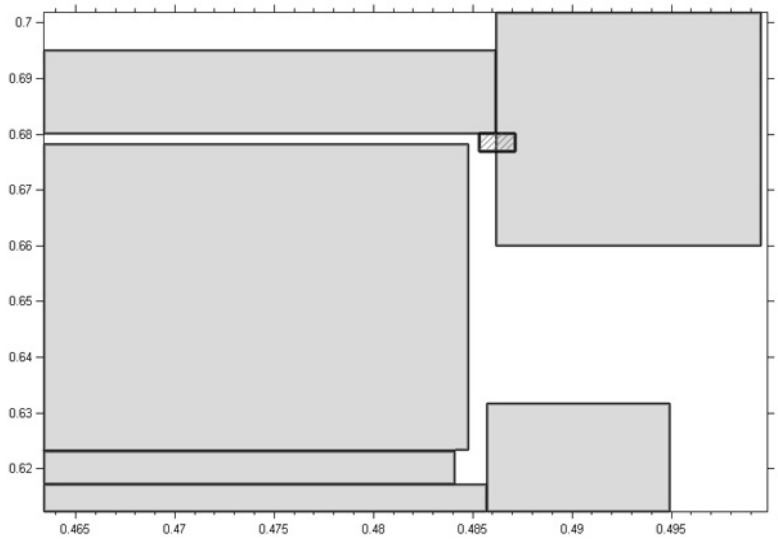
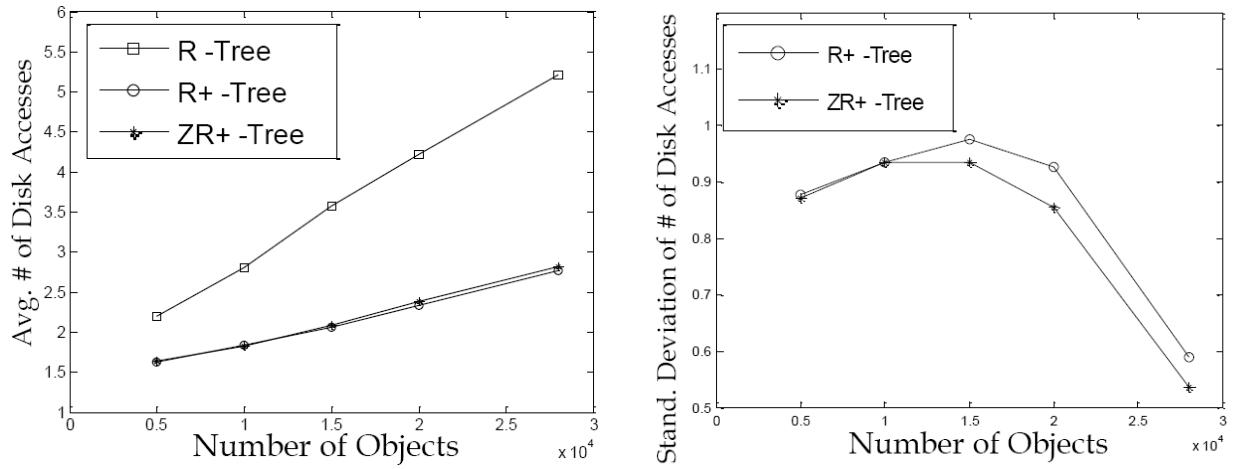
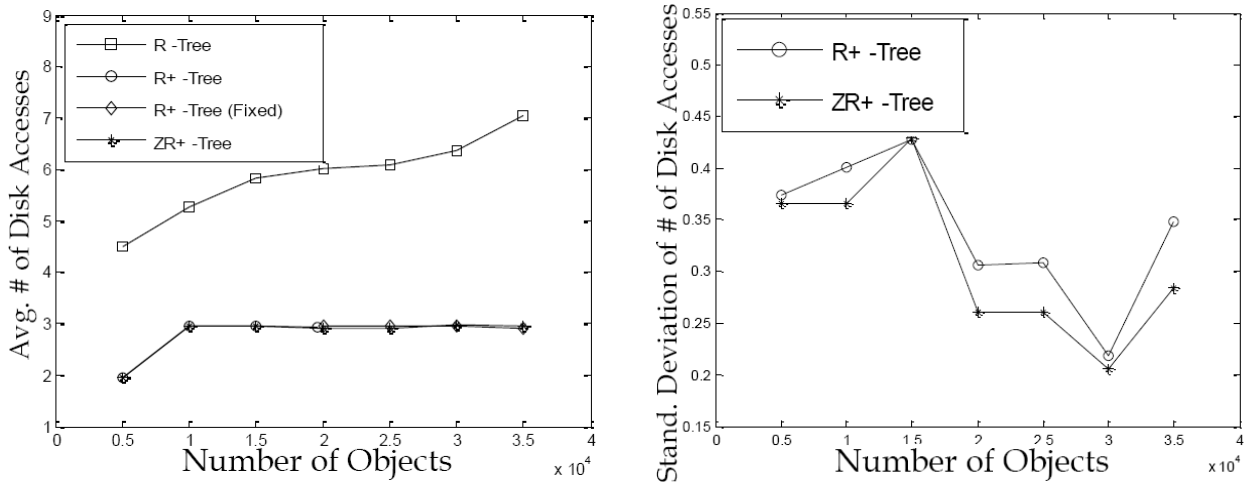


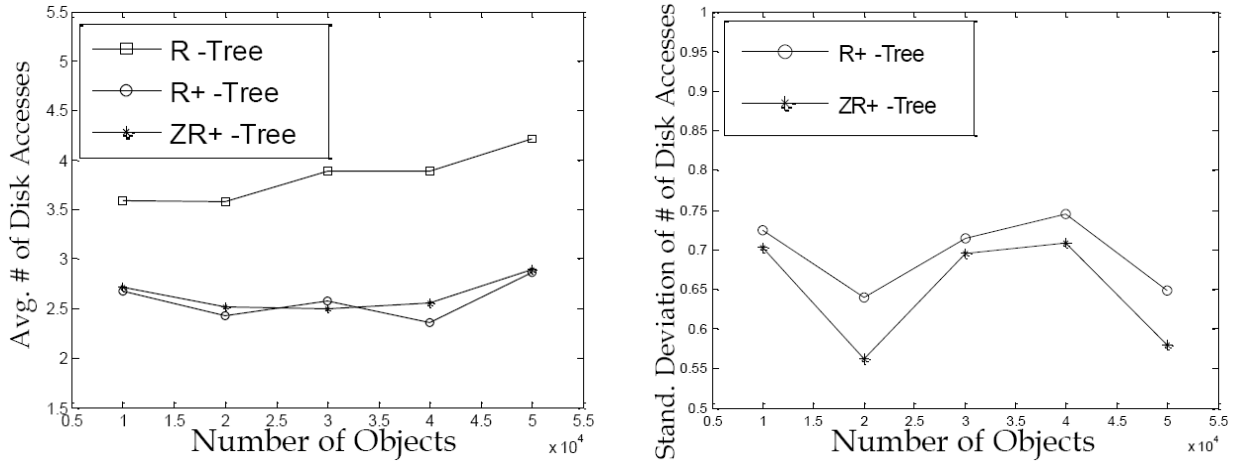
Figure 15. Construction Failure in Building R+-tree on Roads of Long Beach data.



(a) Point Query on Major Roads of Germany



(b) Point Query on Roads of Long Beach County

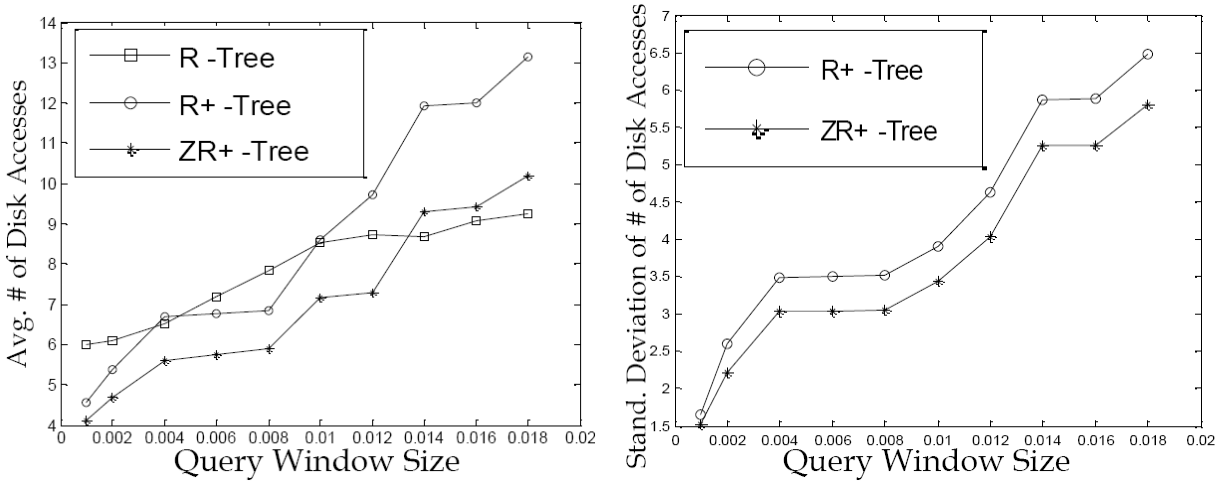


(c) Point Query on Synthetic Data

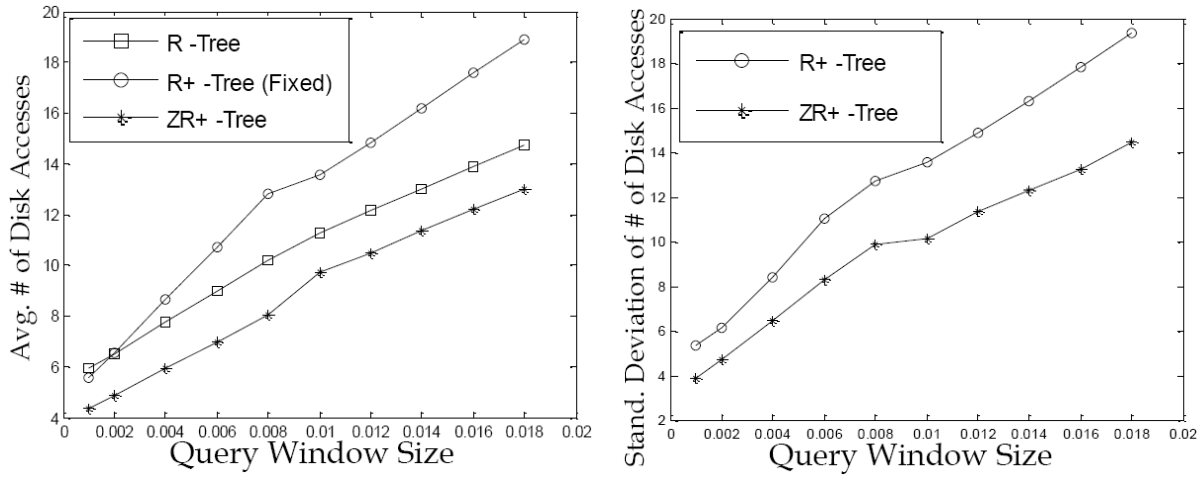
Figure 16. Point Query Performance of R-tree, R+-tree, and ZR+-tree.

3.4.3 Window Query

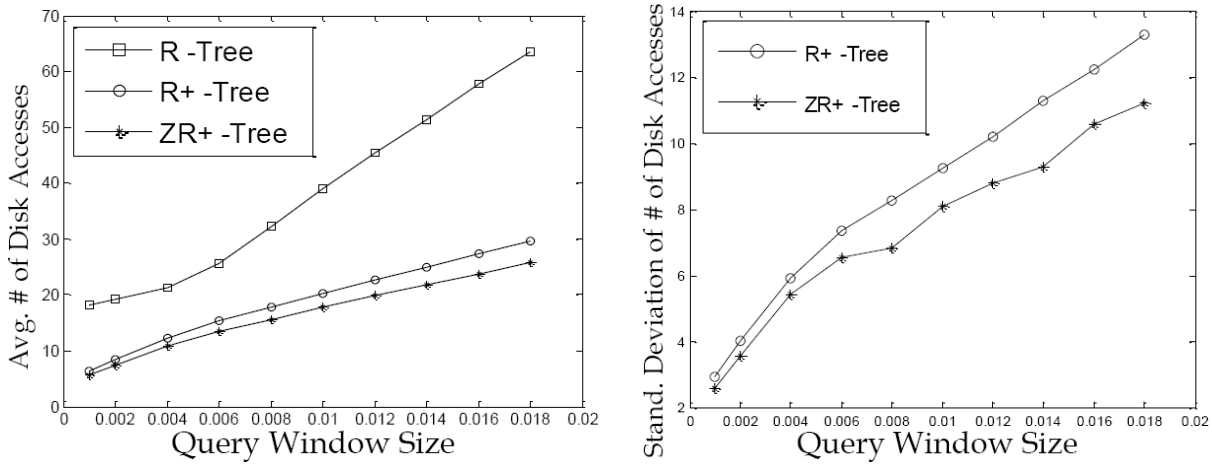
For window queries, the full datasets were used in the experiments (28,014 rectangles for Major German Roads, 34,617 rectangles for Long Beach County Roads, and 50,000 rectangles for the synthetic data). In the left figure in Figure 17 (a), the ZR+-tree has a similar curve of average disk accesses to that of the R+-tree, but the performance is consistently better. It also performed better than the R-tree when the query window size was set to be no larger than 1.2% of the data space. When the window size increases, because the size of the leaf nodes in the ZR+-tree and the R+-tree are usually smaller than the R-tree which allows for overlap among the nodes, window queries in the ZR+-tree and the R+-tree will cover more leaf nodes than in the R-tree, which thus increases the number of disk accesses required. For the same reason, the R+-tree performed worse than the R-tree in Figure 17(b) for query windows larger than 0.2% in terms of disk accesses. In all the three datasets, the performance of the ZR+-tree was generally better than the R-tree and the R+-tree, with the window size varying from 0.1% to 2% of the dataset. The only exception was when the window size was larger than 1.2% in the German Roads dataset. Furthermore, the R+-tree had higher standard deviations than the ZR+-tree when they were using the same query window sizes in the right-hand-side plots in Figure 17 (a), (b) and (c). As in most real applications, the size of the query window would be much smaller than 1% of the whole dataset, and these results show that the ZR+-tree outperforms both the R+-tree and the R-tree for most window queries.



(a) Window Query on Major Roads of Germany



(b) Window Query on Roads of Long Beach County



(c) Window Query on Synthetic Data

Figure 17. Window Query Performance of R-tree, R+-tree, and ZR+-tree.

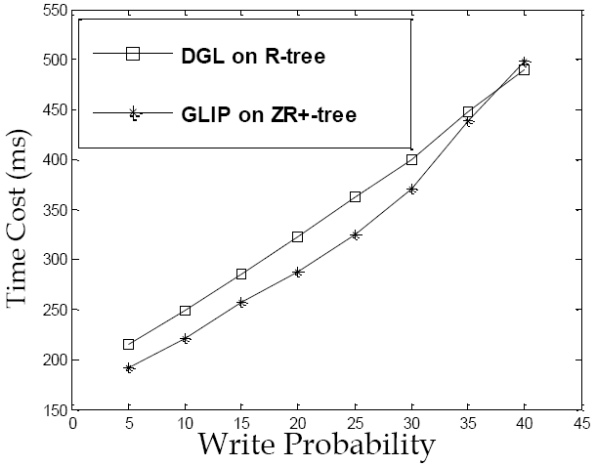
3.4.4 Throughput of Concurrency Control

The performance for concurrent query execution was evaluated both for the R-tree with granular locking and ZR+-tree with the proposed GLIP protocol. In order to compare these two multidimensional access frameworks, two parameters, namely concurrency level and write probability, have been applied to simulate different application environments on the three datasets. Concurrency level is defined as the number of queries to be executed simultaneously, and write probability describes how many queries in the whole simultaneous query set are update queries. The execution time measured in milliseconds was used to represent the throughput of each of the approaches. According to performance analysis, ZR+-tree with concurrency control should perform better than the R-tree with granular locking when the write probability

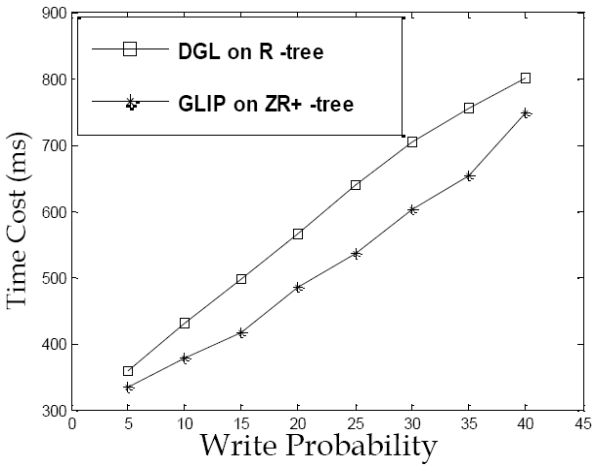
is low. The performance gain comes from not only the outstanding query performance of ZR+-tree, but also the finer granules of the leaf nodes in ZR+-tree, which provides better concurrency. The size of the queries executed was tunable in this set of experiments. The datasets used in these experiments were the same as those used in the query performance experiments, except that the size of the synthetic dataset was reduced to 5,000, in order to assess the throughput in relatively small datasets compared to the real datasets.

Figure 18 shows the execution time costs for the three datasets with a fixed concurrency level and changing write probabilities when the query range was 1% of the data space. The concurrency level was fixed at two levels, 30 and 50 as representative levels, while the write probability was varied from 5% to 40%. The y-axis in these figures shows the time taken to finish these concurrent operations, and the x-axis indicates the portions of update operations in all the concurrent operations in terms of percentage. Both approaches delivered a degrading throughput when the write probability increased. Comparing the performance from the different write probabilities, GLIP on ZR+-tree performed better than granular locking on the R-tree when the write probability was small. When the write probability increased, the throughput of the concurrency control on the R-tree came close to and exceeded that of ZR+-tree. Specifically, when the concurrency level was 30, the throughput of ZR+-tree was better with a write probability lower than 30% in real datasets. When the concurrency level was raised to 50, the concurrency control on ZR+-tree outperformed the R-tree in cases where the write probability was less than 35%. From this set of figures, it can be concluded that in reading-predominant environments, GLIP on ZR+-tree provides better throughput than dynamic granular locking on the R-tree. And this gap usually decreases when the write probability increases.

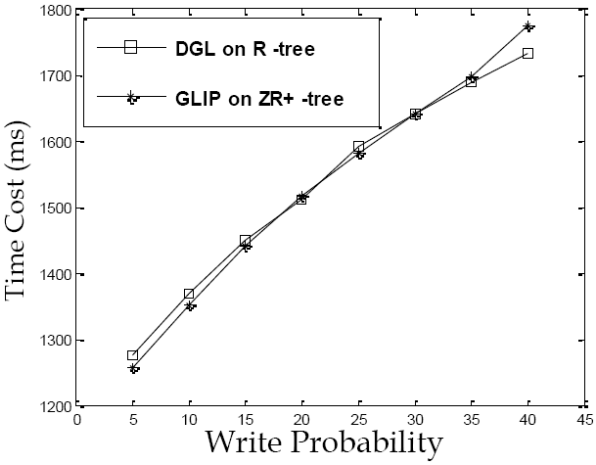
Figure 19 illustrates how the concurrency control protocols performed with fixed write probabilities under different concurrency levels. The y-axis shows the time costs to finish the concurrent operations in terms of milliseconds, and the x-axis represents the number of concurrent operations. The write probabilities were fixed as 10% and 30%, as the representatives to show the trends, while the concurrency level was varied from 10 to 150. In these experiments, the GLIP on the ZR+-tree consistently performed better than or similar to the DGL on the R-tree. When the concurrency level increased, the advantage of the GLIP on the ZR+-tree became more and more significant when compared to the DGL on the R-tree. Observed from the figures, the execution time of the GLIP on the ZR+-tree was significantly lower than the DGL on the R-tree when the concurrency level was more than 50 in the two real datasets and more than 10 in the synthetic dataset, with the write probability 10%. All the figures in Figure 19 show a similar trend, which is that the advantage of the GLIP on the ZR+-tree increases when the number of concurrent operations increases, and that is more significant on evenly distributed datasets, comparing to the DGL on the R-tree.



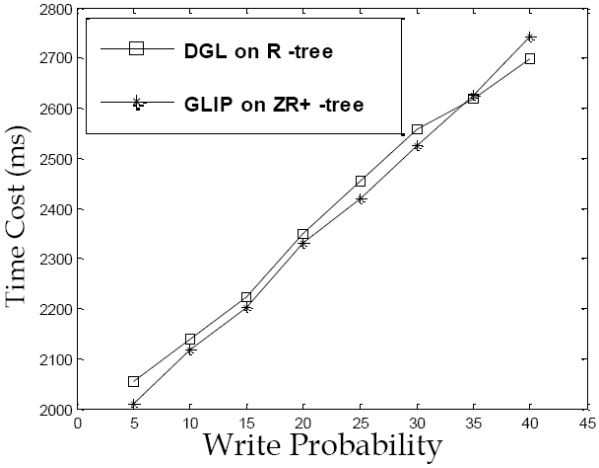
(a) Synthetic Data: Concurrency Level = 30



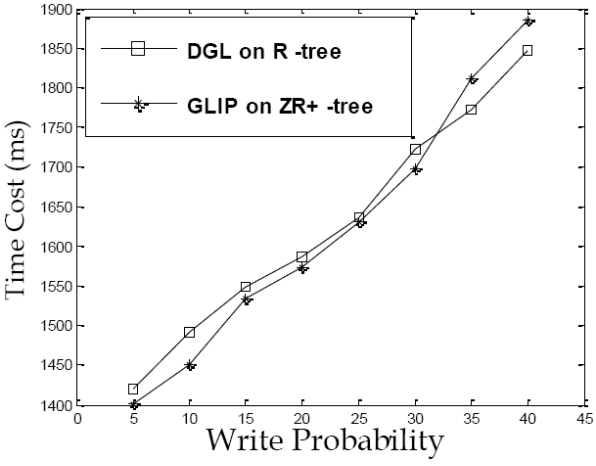
(b) Synthetic Data: Concurrency Level = 50



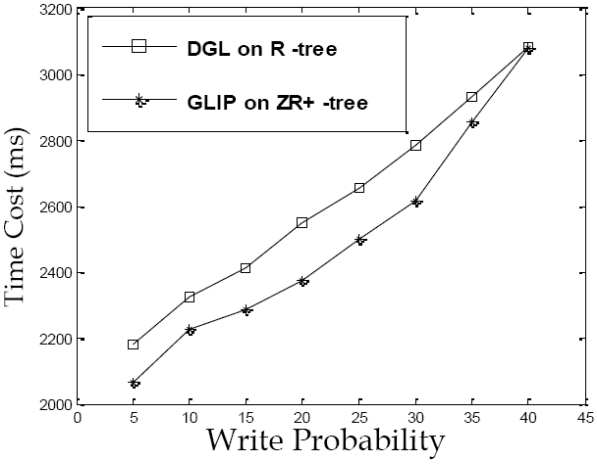
(c) Major Roads of Germany: Concurrency Level = 30



(d) Major Roads of Germany: Concurrency Level = 50

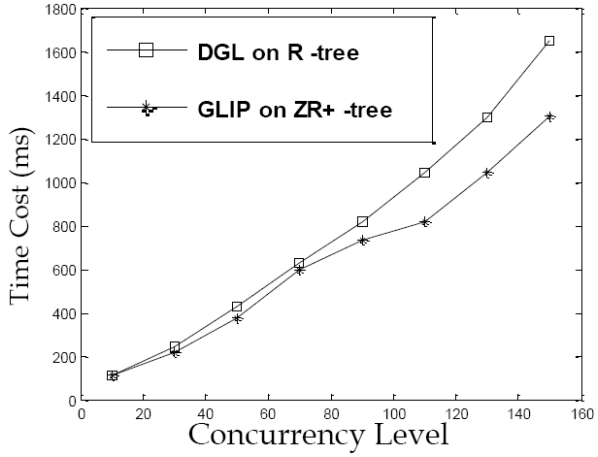


(e) Roads of Long Beach County: Concurrency Level=30

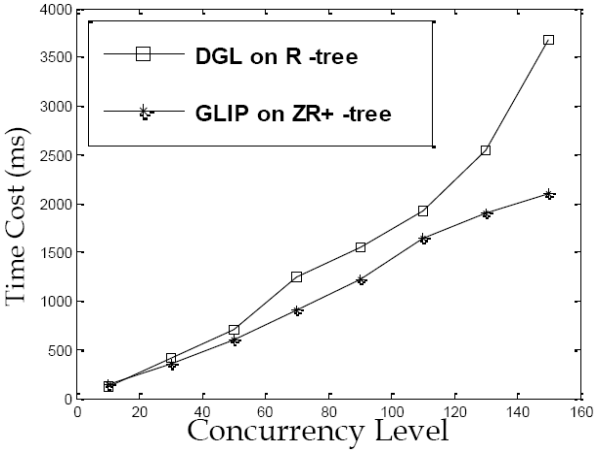


(f) Roads of Long Beach County: Concurrency Level=50

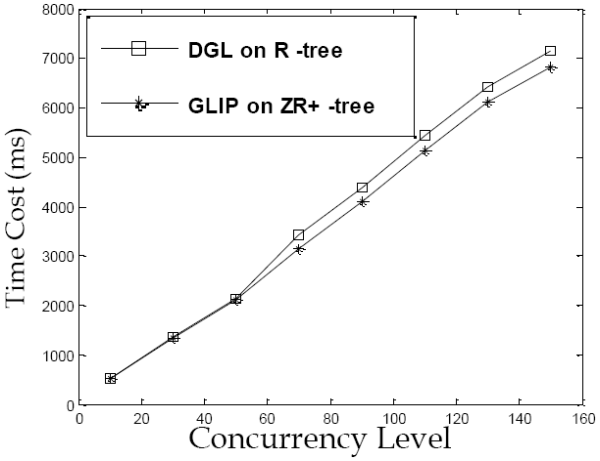
Figure 18. Execution Time for Different Concurrency Levels.



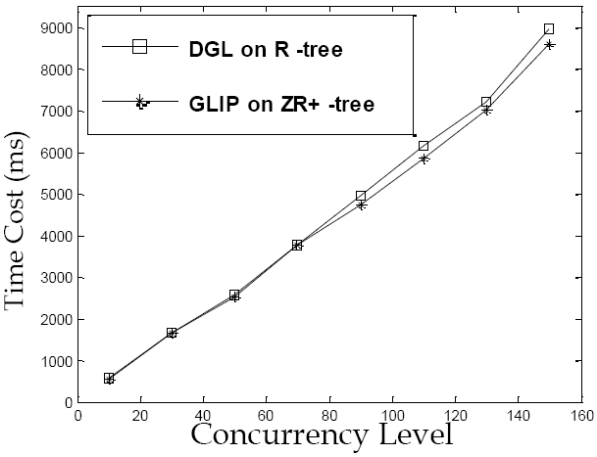
a) Synthetic Data: Write Probability = 10%



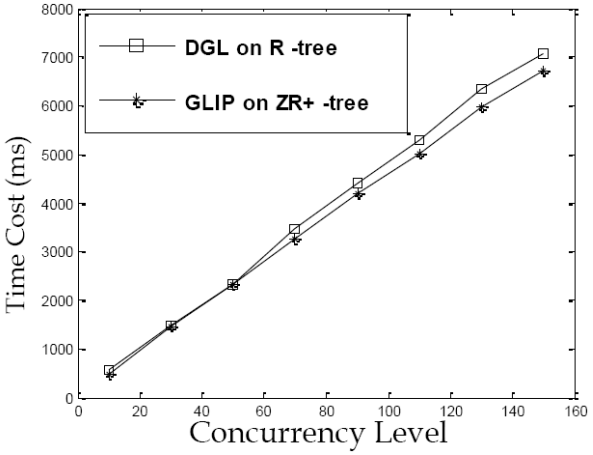
b) Synthetic Data: Write Probability = 30%



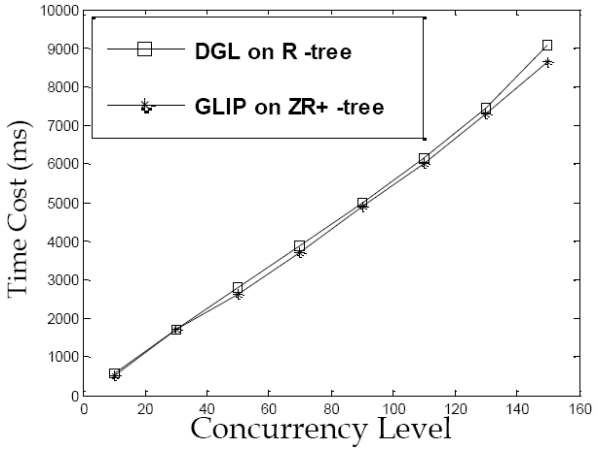
c) Major Roads of Germany: Write Probability = 10%



d) Major Roads of Germany: Write Probability = 30%



e) Roads of Long Beach County: Write Probability=10%



f) Roads of Long Beach County: Write Probability=30%

Figure 19. Execution Time for Different Write Probabilities.

To summarize the experimental results on both query performance and concurrency control throughput, the

ZR+-tree outperforms the R-tree in terms of both point query and window query costs, and outperforms the R+-tree in terms of the I/O cost for window query and the stableness of point query. Comparing the concurrency control protocol, the GLIP on the ZR+-tree performs better than dynamic granular locking on the R-tree especially with high concurrency and low write probability. It is particularly suited to applications that access multi-dimensional data with high concurrency and low write probability.

3.5 Conclusion

This chapter proposes a new spatial indexing approach, ZR+-tree, for use with a corresponding concurrency control strategy, GLIP. The ZR+-tree is a variation of the R-tree, which segments the objects to make each piece of them fully covered by a leaf node. This clipping-object design will provide a better structure than the R+-tree. Furthermore, several structural limitations of the R+-tree are overcome in the ZR+-tree by the non-overlap clipping and clustering-based re-insert. With these techniques, the ZR+-tree can provide better query performance in search-predominant applications than other major R-tree variants. The concurrency control protocol, GLIP, is the first concurrency control mechanism designed for the R+-tree and its variants. Experiments on tree construction, query and concurrent execution were conducted on both real and synthetic datasets, and the results obtained support the soundness and comprehensive nature of the new design.

Future efforts could focus on extending the ZR+-tree and the GLIP to perform complex operations, such as spatial joins and range aggregation queries, to support attractive real-world applications.

Chapter 4. CONCURRENCY CONTROL ON CONTINUOUS QUERIES

In this chapter, an efficient concurrent continuous query processing approach, Concurrency Control on Continuous Queries (C^3), is proposed to handle moving objects in multi-user environments. C^3 supports efficient concurrent location updates and continuous queries on an access framework that fuses scalable continuous query processing methods and state-of-the-art lazy update techniques. This proposed concurrency control protocol, equipped with intra-index and inter-index protection on a generalized indexing structure, assures serializable isolation, consistency, and deadlock-freedom for this continuous query processing framework.

4.1 Preliminaries

This work provides a solution for concurrent continuous range queries on multi-dimensional moving object databases. In this problem, each range query keeps monitoring a rectangular area and refreshing the query results based on the object movement and query movement. Concurrent operations supported in the system are object location update, query location update, and query report. An **object location update** operation inputs both the old position and new location of a *d*-dimensional data point, and updates the object database, object index, and query results. Similarly, a **query location update** operation inputs both the old position and new position of a *d*-dimensional window, and updates the query database, query index, and query results. A **query report** returns a set of data objects that currently overlaps with a given range at the time the report is finished. These operations should not interfere with each other, and the outputs of the query report operation should reflect the current consistent state of the database.

The proposed C^3 works for the R-tree-based spatial access methods with lazy update. This generalized access method has the features of both lazy group update and update memo techniques, so the proposed protocol can work on either variant of R-trees. R+-trees and ZR+-trees are not recommended for using in C^3, because they perform more expensive updates than the R-tree. In order to focus on the concurrency control protocol, the access method is generalized by only maintaining the current locations of the queries and objects. However, it is convenient to extend the concurrency control protocol to other motion-sensitive indices such as MAI [14].

To specify the problem to be solved, several assumptions for the application environment have to be made.

1. Point objects: Moving objects are represented by spatial points; each object may report its new location

to the database at any time if moved.

2. Window queries: Moving queries are represented by their query windows (spatial boxes); each query may periodically report its new query window to the database at any time if moved.
3. Lock manager: There exists a lock manager to support different lock types and maintain all the locks. In addition, there is a system counter to assign a globally unique timestamp to each operation.

The above assumptions are reasonable in many applications. With these assumptions, an efficient concurrent spatial access framework for continuous queries can be designed.

4.1.1 Access Framework

The proposed concurrency control protocol is based on a generalized spatial access framework that combines existing methods for frequent update and continuous query processing, so that it can be easily applied to the frameworks that employ any of these techniques. The generalized access method applies two R-trees with lazy group update for insert operation and update memo for delete operation, one for moving objects (*O-tree*, as shown in Figure 20) and another for moving queries (*Q-tree*, as shown in Figure 21). The indexing structures of *O-tree* and *Q-tree* are exactly the same.

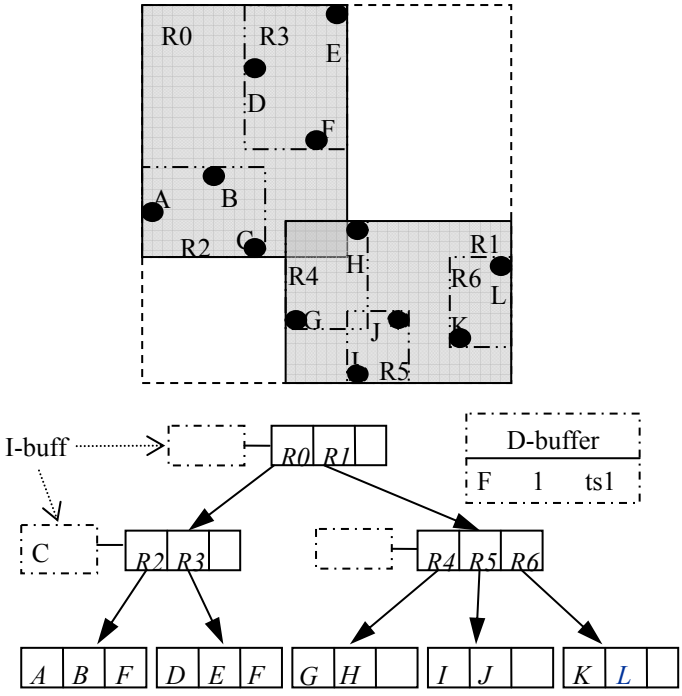


Figure 20. An Example of O-tree with I-buffer and D-buffer.

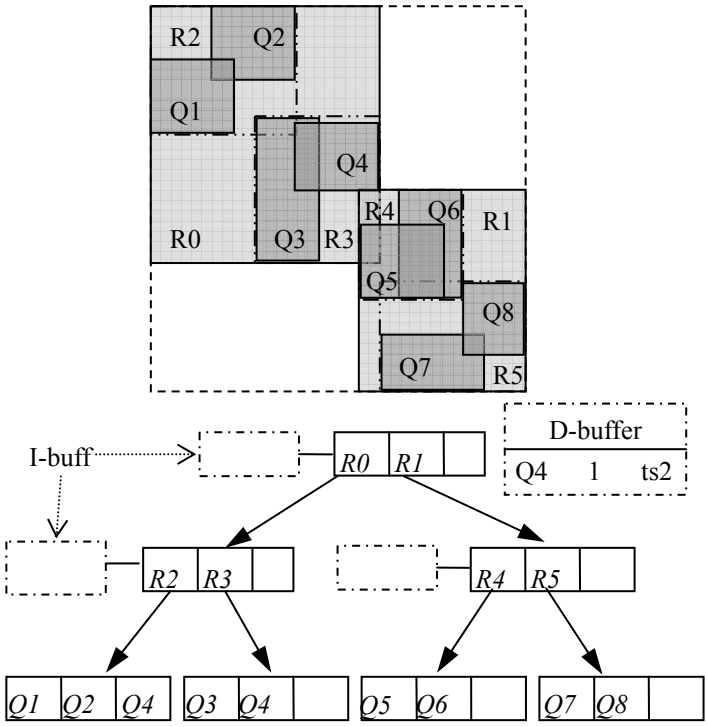


Figure 21. An Example of Q-tree with I-buffer and D-buffer.

The lazy group update requires one insert buffer *I-buffer* (dashed boxes connected to each non-leaf node in Figure 20) for each non-leaf node of the R-trees. The *I-buffer* temporally stores the inserted objects or queries on the appropriate level; and if full, pushes the largest group of inserted objects or queries down to the particular *I-buffer* on the next level or to the leaf node [19]. Each entry of an *I-buffer* has the form (*Oid/Qid*, *MBR*, *target_child*, *timestamp*).

On the other hand, the update memo needs a delete buffer *D-buffer* (dashed boxes beside the tree in Figure 20) for each R-tree. The *D-buffer* caches the delete operations by recording the object/query *IDs*, the number of obsolete records for that *ID*, and the most recent timestamp, and removes the obsolete records in leaf nodes when processing garbage collection [17]. Each entry of a *D-buffer* has the form (*Oid/Qid*, *#_obsolete*, *timestamp*).

On either *O-tree* or *Q-tree*, a **range search** needs to traverse the tree with *I-buffer* to find records overlapping with the search range. In this structure, search results can appear not only on leaf nodes, but also in *I-buffer*s. Before outputting the objects, *D-buffer* has to be checked to remove the obsolete objects from the results. An **insert** operation on a tree first tries to insert the given item into an *I-buffer* whose hosting node is chosen to include this item based on the R-tree algorithm. If the target *I-buffer* is full, it will be re-organized by: 1) removing obsolete items by checking *D-buffer*; 2) executing lazy group update to

push some items to an *I-buffer* in the next level or to a leaf node. A **delete** operation on either of these indexing trees only needs to add this delete record to *D-buffer* with the current timestamp. *D-buffer* can be cleaned by going through the leaf nodes to remove obsolete items.

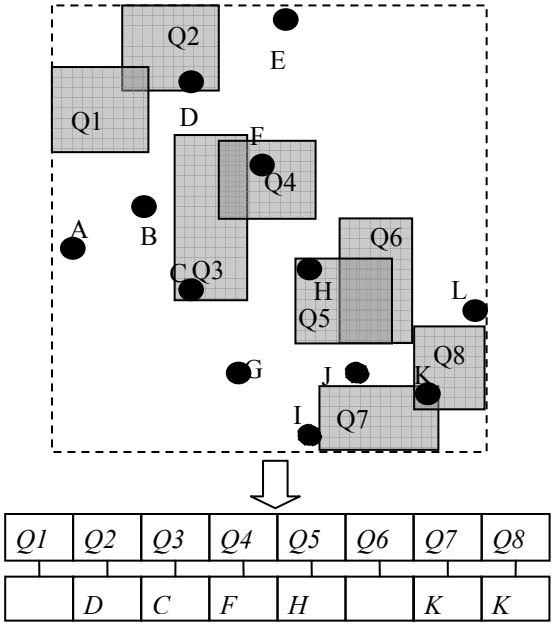


Figure 22. Q-result for Objects in Figure 20 and Queries in Figure 21.

In addition to *O-tree* and *Q-tree*, there is a hash-based array, *Q-result* (as shown in Figure 22), to store all the results for continuous range queries. The *Q-result* is hashed by query *ID*s, and each particular entry corresponds to a continuous query. Each entry of *Q-result* is in the form of (*Qid*, *obj_list*), corresponding to a query *ID* and the list of objects covered by the query. The *obj_list* also contains the timestamp of each object in the list. The example of *Q-result* in Figure 22 reflects the dataset and query set illustrated in Figure 20 and Figure 21 accordingly.

The continuous queries on this indexing framework are processed via three operations, query report, object location update, and query location update. The details of these operations are discussed in Section 4.2.

4.1.2 Concurrency Control Protocol

Continuous query processing requires an appropriate concurrency control protocol to ensure the correct results while objects and queries are moving. In the scenario in Figure 2, inconsistent results are caused by incorrect processing sequences. Suppose each movement can be divided into three 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 containing … →*A.D*→*Q.R*→*A.I*→*qR*→*A.R*→... will output *null* as the results of *Q* at time t_2, because *A* disappears in the database when *Q* updates its results, and no further update occurs before the query report for 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*→…. 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* has any chance to re-evaluate 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. 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.

In the proposed protocol, the lockable items include leaf nodes and external granules of nodes (defined in DGL [9]) on both trees, entries in each *D-buffer*, and entries in *Q-result*. *I-buffer*s in both trees do not need specific locks, because they are always attached to certain internal nodes which can be locked by the external granule. Following the convention of lock-coupling approaches [9], the types of locks that are utilized in the proposed protocol include *S (shared lock), X (exclusive lock), IX (Intention to set X locks), IS (Intention to set S locks)*, and *SIX (Union of S and IX lock)*. The compatibility matrix of the lock types is listed in Table 5.

Table 5. Lock Type Compatibility Matrix.

Lock Type	IS	IX	S	SIX	X
IS	√	√	√	√	
IX	√	√			
S	√		√		
SIX	√				
X					

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

The proposed concurrency control protocol supports serializable isolation by providing protection from the

following issues.

- Inconsistency within each indexing structure;
- Inconsistency among *D-buffers*, trees, and *Q-result*;
- Deadlock caused by accessing multiple indices.

By solving the above issues, C^3 completely protects the whole framework for continuous query processing as a consistently integrated structure that outputs valid results.

4.2 Concurrent Operations

The proposed concurrency control protocol supports concurrent operations on the generalized access framework, including query report, object location update, and continuous query location update. These operations can be simultaneously processed without interfering with each other. The concurrent operations are described in detail in the following.

4.2.1 Query Report

The query report operation is to read the moving objects covered by a continuous query. The operation takes a query *ID* as input and outputs a set of object *ID*s. In the proposed indexing framework, it needs to read the particular entry in *Q-result* and validate the results by looking up in the *D-buffer* of *O-tree*, because *Q-result* is a hashed array that may contain obsolete objects. In detail, with C^3, the query report first puts *S-lock* on the entry with given query *ID* in *Q-result*, and reads the corresponding *obj_list*. It then requests *S-lock*s on the *D-buffer* of *O-tree* for all the objects that are contained in *obj_list*, checks and removes the obsolete objects from *obj_list*. At last, this operation returns the remaining objects and releases all the *S-lock*s.

As an example, based on the *Q-result* in Figure 22, a query report with the input *Q5* requests an *S-lock* on the entry *Q5* in *Q-result* and then finds the object *H* in the entry. An *S-lock* is then requested on the entry of *H* in *D_buffer*. Because there is no deletion record for *H* in the *D-buffer* of *O-tree*, the algorithm simply releases all the *S-lock*s and returns *H* as final result. The algorithm of the query report is illustrated in Algorithm 6. This operation can be requested by a client at any time or triggered by the corresponding updates of *Q-result*.

```
Algorithm Query_Report
Input: Qid: Query ID
Output: S: Set of Objects

S-lock(Q-result.entry[Qid]);
S = Q-result.entry[Qid].obj_list;
For each pair (Oid, ts) in S
      S-lock(O-tree.D-buffer.entry[Oid]);
      If Oid in O-tree.D-buffer
            If O-tree.D-buffer.entry[Oid].ts > ts
                  S = S – (Oid, ts);
      Unlock(O-tree.D-buffer.entry[Oid]);
Unlock Q-result.entry[Qid];
Return S;
```

Algorithm 6. Query Report.

4.2.2 Object Location Update

The object location update operation updates the location of the object, as well as the results of affected queries. It takes the new location of the object and performs a lazy update on *O-tree* and another update on *Q-result*. There are three phases in this operation, location update on *O-tree*, point search on *Q-tree* and update *Q-result*, as shown in Algorithm 7. The details of these phases are presented as follows.

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

Algorithm Object_Location_Update

Input: Oid: Object ID, loc_old: Old Location of Object, loc_new: New Location of Object, O-tree: Index Tree of Objects, Q-tree: Index Tree of Queries, Q-result: Result Sets for Queries
Output: Nil

```
ts = get_timestamp();
QList = Nil; //set of queries that cover the point

//Phase 1. O-tree location update
//delete old location
X-lock(O-tree. D-buffer.entry[Oid]; //avoid operations on same object
O-tree.D-buffer.Add(Oid, #_obsolete, ts);
//insert new location
X-lock(O-tree.root.ext);
curNode = O-tree.root;
Add (Oid, loc_new, ts) to curNode.I-buffer;
Enlarge curNode.MBR to cover loc_new;
While curNode.I-buffer is full and not curNode.isLeaf
      nextNode = curNode.I-buffer.childForUpdate();
      X-lock(nextNode.ext);
      curNode.I-buffer.groupUpdate(); //push the largest group of contents to its target child
      Unlock(curNode.ext);
      curNode = nextNode;
Unlock(curNode.ext);

//Phase 2. Q-tree point search
S-lock(Q-tree.root.ext);
curNode = Q-tree.root;
While not curNode.isLeaf
      QListTmp = curNode. I-buffer.find(loc_new);
      If (S-lock(Q-tree. D-buffer.entry[QListTmp], Conditional)) //conditional lock, true if the lock is granted
          X-lock(Q-result.entry[QListTmp]); //avoid operations on same query
          QList.add(QListTmp);
      nextNode = curNode.findEntry(loc_new);
      S-lock(nextNode.ext);
      Unlock(curNode.ext);
      curNode = nextNode;
QListTmp = curNode.find(loc_new);
If(S-lock(Q-tree. D-buffer.entry[QListTmp], Conditional)) //conditional lock
      X-Lock(Q-result.entry[QListTmp]); //avoid operations on same query
      QList.add(QListTmp);
Unlock(curNode.ext);
QList = Q-tree. D-buffer.filter(QList); //remove obsolete queries in QList
Unlock(Q-tree. D-buffer.entry[QList]);

//Phase 3. Q-result Update
For each Qid in QList
      Update Q-result.entry[Qid].obj_list by adding (Oid, ts);
      Unlock(Q-result.entry[Qid]);
UnLock(O-tree. D-buffer.entry[Oid]);
Return;
```

Algorithm 7. Object Location Update.

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

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

An example based on the objects and queries in Figure 20 and Figure 21 can demonstrate this object location update. Suppose the object *C* is moving into the region of node *R4* and also covered by *Q5*. The system first locks the entry of *C* in the *D-buffer* of *O-tree*, although there is no record for *C* yet. Then an entry (*C*, *1*, *ts*) is inserted into the *D-buffer*. *C* is then inserted into the *I-buffer* of root in *O-tree* with *ts* following the algorithm. In phase 2, a point search is performed on *Q-tree* using the new location of *C*, and *Q5* is retrieved. The entry of *Q5* in the *D-buffer* of *Q-tree* is *S-lock*ed, and the entry of *Q5* in *Q-result* is *X-lock*ed. After checking *D-buffer*, *Q5* is confirmed as the real affected query by *C*. The *obj_list* of *Q5* is now updated to contain *C* and *H*. Finally, the locks on the entry *C* in *D-buffer* and on the entry *Q5* in *Q-result* are released.

4.2.3 Query Location Update

The query location update operation handles the location change of a query. This change could be on the location or the size of the query window, or both. This operation takes the new search window of a given query as input, and updates the *Q-tree* as well as the *Q-result* accordingly. Similar to the object location update operation, its algorithm consists of three phases, namely, location update on *Q-tree*, range search on

O-tree, and update *Q-result*. Timestamp *ts* is assigned at the beginning of the processing, so that the lazy update can have a sequential order for the update records.

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

Phase 2 - *O-tree* range search: It queries the new query window on *O-tree* to retrieve all the objects that are covered. This range search scans the nodes as well as their *I-buffers* on the traversal path, and requests *S-lock*s for the covered granules and the corresponding entries in the *D-buffer* of *O-tree*. Note that the *S-lock*s on the *D-buffer* of *O-tree* are unconditional, which means if these objects are locked with *X-lock* by other operations, this query location update will give up those objects. This is because if an object is locked by object location update, it will be added to the corresponding queries according to its new location. So there is no need to include this object in the *Q-result* update. This unconditional lock also can prevent the deadlock between object location update and query location update.

Phase 3 - *Q-result* update: It replaces particular entry of *Q-result* with the new set of objects that have been found as the results. The *X-lock* on the entry of *Q-result* and the *S-lock*s on the corresponding entries of the *D-buffer* on *O-tree* are released once the update is finished. The details of the algorithm are shown in Algorithm 8.

The algorithm of query location update can be illustrated using an example based on the objects and queries in Figure 20 and Figure 21. Assume the query *Q8* is moving upward within *R1* and now covers object *L* with the new query window. It still can be included in the extended leaf node *R5* according to the design of the R-tree. The system first locks the entry of *Q8* in the *D-buffer* of *Q-tree*, although there is no record for *Q8* yet. Then an entry (*Q8*, *1*, *ts*) is inserted into the *D-buffer*. *Q8* is then inserted into the *I-buffer* of the root in *Q-tree* following the algorithm. An *X-lock* is requested on the entry for *Q8* in *Q-result* before the lock on the *D-buffer* of *Q-tree* is released. In phase 2, a window search is performed on *O-tree* using the new query window of *Q8*, and the object *L* is retrieved. The entry of *L* in the *D-buffer* of *O-tree* is *S-lock*ed

before checking its validity. After *L* is confirmed as the real covered object by *Q8,* the *obj_list* of *Q8* is now replaced by *L*. Finally, the locks on entry *L* in *D-buffer* and on the entry *Q8* in *Q-result* are released.

Algorithm Query_Location_Update

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

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

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

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

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

Algorithm 8. Query Location Update.

4.2.4 Garbage Cleaning

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

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

Using the *O-tree* and *D-buffer* in Figure 20 as an example, when the leaf node *R2* requests a *D-buffer* clean, *X-lock*s are placed on the leaf node *R2* and entries *A*, *B* and *F* in the *D-buffer*. After checking the timestamps in deletion records, the object *F* is found as obsolete. The entry *F* is then removed from the leaf node *R2*. Since the number of obsolete records of the entry *F* in the *D-buffer* was 1, after removing *F* from *R2*, this entry in the *D-buffer* can be removed too.

4.3 Correctness

The proposed concurrency control protocol C^3 assures serializable isolation, consistency, and deadlock-freedom on the generalized access framework. **Serializable isolation** means the results of any set of concurrent operations are equal to those from the sequential processing of the same set of operations. **Consistency** refers to the feature that the results always reflect the current committed status of the database. **Deadlock-freedom** means any combination of the concurrent operations will not cause any deadlock. The correctness of this concurrency control protocol is discussed in detail as follows by analyzing the lock durations for each operation illustrated in Figure 23.

Figure 23 abstracts the order and duration of the locks requested in each operation, including object location update, query location update, query report, and garbage cleaning in *D-buffer*. The garbage

cleaning in *I-buffer* only processes inside an R-tree, so it won't cause any correctness issue with the inter-structure operations. The abbreviations in Figure 23 indicate the locks on different structures. The items *ON* and *QN* are the locks inside the R-trees, while the items *OD*, *QD* and *QR* are the locks that take charge of the inter-structure protection on the operations. Objects and their corresponding *O-tree* nodes are locked in *ON* and *OD*, while queries and their corresponding *Q-tree* nodes are protected in *QN*, *QD* and *QR*. The horizontal span of each bar in the figure represents the time period that the locks are granted on the corresponding structure. Based on the algorithms, search operations request *S-lock*s, and update operations request *X-lock*s. The object location update and query location update will not request *S-lock*s on the same substructure according to the protocol. Query report only requests *S-lock*s, while garbage cleaning in *D-buffer* has to place *X-lock*s. Among these locks, *ON* and *QN* are gradually requested by traversing the tree; the other locks for each bar are granted at almost the same time.

Serializable isolation: The proof of serializable isolation contains two parts, serializable isolation on the single tree and among *O-tree*, *Q-tree*, and *Q-result*. The serializable isolation on a single R-tree has been proved [9]. A similar proof can show that *O-tree* and *Q-tree* are internally serializable, because the sub-operations on each single tree (*ON* and *QN*) are protected like on an R-tree, except that the locks on tree nodes cover the associated *I-buffer*s.

On the other hand, the serializable isolation among the *O-tree*, *Q-tree*, and *Q-result* can be proved based on the theory that a group of transactions are serializable if and only if their conflict graph has no cycles [5]. We prove this in the following lemma.

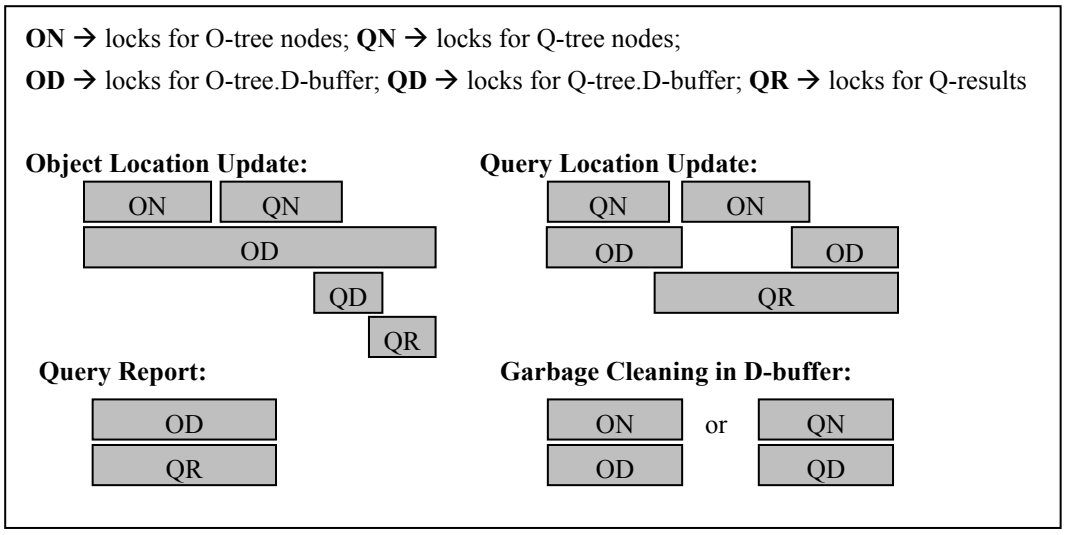


Figure 23. Lock Durations for Concurrent Operations

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

Proof: We prove this lemma using induction. Given that any sub-operations that involve a single index tree are serializable, because of the conditional lock applied in the algorithm, the sub-operations on index tree nodes and *I-buffer*s are serializable to each other. Therefore, we only need to consider the sub-operations corresponding to *OD*, *QD*, and *QR*. Figure 24 illustrates the major steps of the proof.

Step1 - acyclic between two operations: to prove a conflict graph with any two operations in this framework is acyclic. Based on the lock durations illustrated in Figure 23, obviously a conflict graph with any two same operations is acyclic. Considering two different operations, a query report or a garbage clean cannot cause a cycle in a conflict graph with another operation, because the locks requested by a query report or a garbage clean are maintained until its commit point. Based on Figure 23, an *OLU* and a *QLU* can cause a potential cycle in a conflict graph, because these two operations may involve the same object and query. However, because of the conditional locks applied in the algorithms, if an *OLU* realizes that the query affected by the object is locked by a *QLU*, it will not access that query or update the corresponding query result. The same rule applies to the potential conflict on objects in a *QLU*. Because if two operations conflict on objects, they must conflict on queries too, no edges for object locations can be drawn between an *OLU* and a *QLU*. Therefore, in a conflict graph that contains an *OLU* and a *QLU*, the only edge, if it exists, can be drawn either from the *OLU* to the *QLU* or from the *QLU* to the *OLU* for query locations and query results. In other words, no cycle could occur in a conflict graph that consists of two operations.

Step2 – acyclic among n operations: to prove given that a conflict graph with n operations ($OP_1, \ldots OP_n$) is acyclic, the conflict graph with operations ($OP_1, \ldots, OP_n, OP_{n+1}$) is also acyclic. Based on the proof in step 1, if OP_{n+1} is a query report or garbage clean, it will not cause any new edges in the graph.

If OP_{n+1} is an *OLU*, a possible edge from OP_i ($1<=i<=n$) to OP_{n+1} can be drawn for query results if OP_i is another *OLU*, or drawn for query locations and query results if OP_i is a *QLU*. Similarly, a possible edge from OP_{n+1} to OP_j ($1<=j<=n, j!=i$) can be drawn for query results if OP_j is another *OLU*, or drawn for query locations and query results if OP_j is a *QLU*. Now we prove there is no path from OP_j to OP_i by contradiction. Assume there exists any path P_{ji} from OP_j to OP_i, because the locks on one lockable structure are granted at the same time, P_{ji} cannot contain any edge drawn for query locations and query results.

However, based on the analysis in step 1, the edges between any two operations only can be query locations and query results. This contradiction shows that the existence of P_{ji} is impossible. Therefore, in case OP_{n+1} is an *OLU*, the conflict graph with operations (OP_1, ..., OP_n, OP_{n+1}) is acyclic.

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

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

Q.E.D..

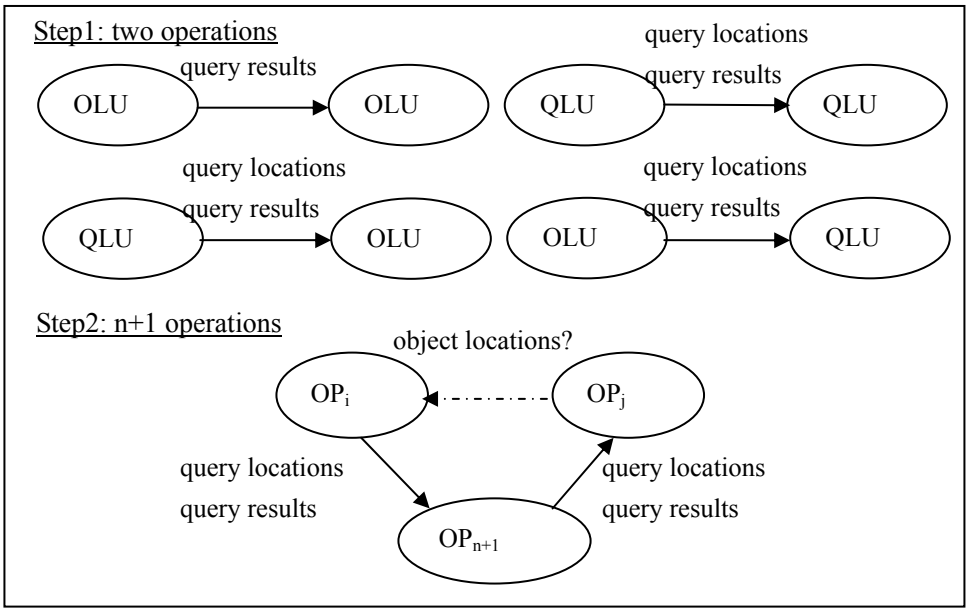


Figure 24. Conflict Graphs for Two Operations and n+1 Operations.

Consistency: For either *O-tree* or *Q-tree*, the DGL approach (*ON* and *QN* in Object Location Update and Query Location Update) has been proved to protect the consistency on the R-tree. Furthermore, from the

above serializable isolation analysis, each proposed operation keeps locking its target items (object/query) throughout the process, which ensures that the intermediate status between any two phases will not be accessed by other operations. Because the query report locks the query (*QR* in Query Report) and objects (*OD* in Query Report) from initiation to termination, only the results of all the operations committed before its initiation will be accessed. This guarantees the continuous query results are consistent with the current database.

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

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

Proof: Because query reports and garbage cleans only request locks at the beginning and release them at the commit point, these operations do not cause any deadlock with any other operations. We discuss *OLU* and *QLU* by observing the lock durations in Figure 23, from the aspects of accessing objects, queries, and objects and queries.

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

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

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

eliminate the conflict. Therefore, the possible deadlocks caused by accessing two trees in opposite orders are prevented by the conditional locks that cede the conflicted objects or queries.

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

Q.E.D..

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

4.4 Experiments

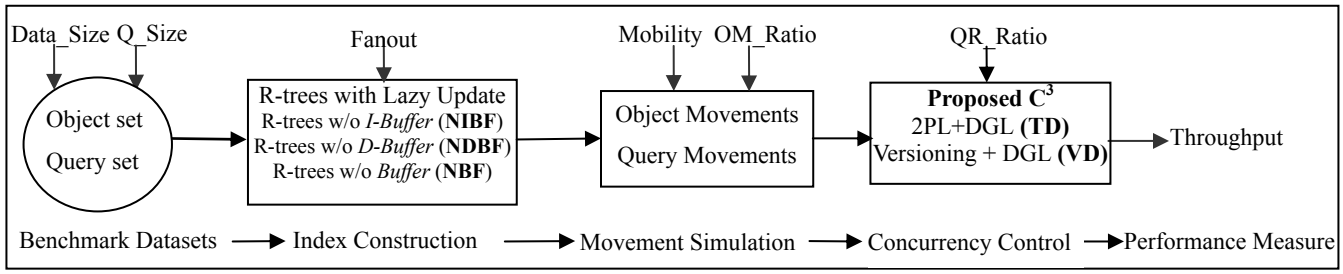


Figure 25. Experiment Flow.

To evaluate the performance of the proposed framework, experiments on benchmark datasets have been conducted by measuring throughput (number of concurrent operations being processed in a second). The design of experiments is illustrated in Figure 25. The benchmark datasets were generated by a network-based moving objects generator [67] using the road network of the City of Oldenburg, as shown in Figure 26. We set three classes of moving objects and queries to represent vehicles, bicycles, and pedestrians. Half of the initial moving objects generated by the generator were used as moving objects, and the second half of the objects were expanded as range queries. Based on the moving object set and moving query set, two 3-level R-trees were constructed with a fanout of 100. Meanwhile, the object movements simulated by the generator were translated into object location updates and query updates. These location updates and a set of random query report operations were sent to the system with C^3 as a multi-thread batch job. The overall execution time for each batch job was collected, and the system throughput was calculated by averaging the throughput of twenty batches of concurrent operations.

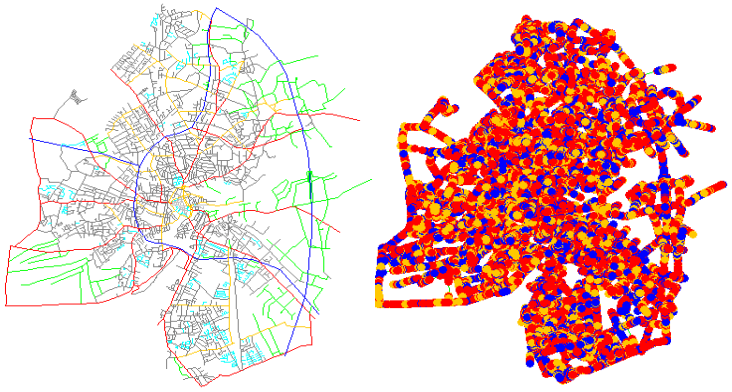


Figure 26. Road Network of Oldenburg and Dataset.

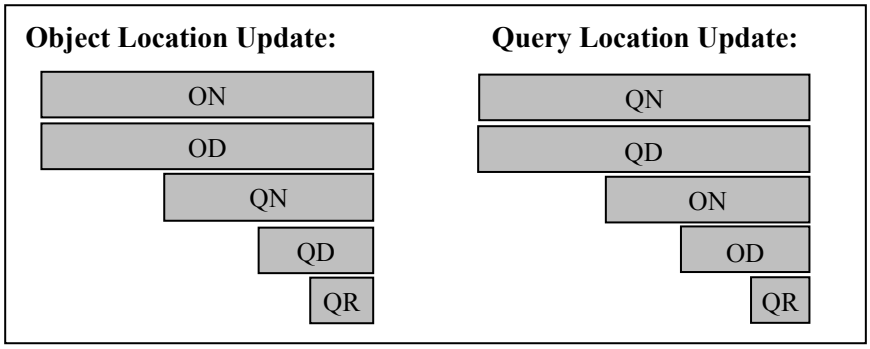


Figure 27. Lock Durations for TD.

An approach that applies a record-oriented two-phase lock transaction management approach, integrated with DGL on R-tree [9] for concurrent operations, namely TD, has been implemented. Another approach integrates a record-oriented versioning approach with DGL, namely VD, has also been developed. TD and VD follow the continuous query processing approach in C^3, except that the operations in TD/VD are executed using the 2-phase locking/versioning strategy among indices and DGL within the R-trees. The lock durations of object location update and query location update in TD are illustrated in Figure 27. It inherits the complete indexing framework from C^3, including O-tree, Q-tree, D-buffer, I-buffer, and Q-result. Therefore, its number of I/O accesses is as optimal as C3. Similarly, VD follows the same query processing algorithms as C^3, except it requires redo operations when conflictions are detected. The conditional locks requested in location updates result in less redo operations in VD than pure versioning protocols, because they allow the operations continue to commit. In other words, TD and VD are both advanced approach for continuous query processing, which can achieve the same performance as C^3 in single-user environments. Because there is no existing concurrent continuous query processing approach in literature, TD and VD are appropriate baselines with serializability for comparison. In addition, simplified

versions of C^3 without operation buffers have been developed to evaluate the impact of the I-buffer and D-buffer. Specifically, three versions, including C^3 without I-buffer (NIBF), C^3 without D-buffer (NDBF), and C^3 without any buffer (NBF), were adopted for comparison.

In the experiments, five parameters were varied to simulate different application scenarios and to demonstrate their respective impacts. These parameters are listed as follows, and their default settings and ranges are listed in Table 6.

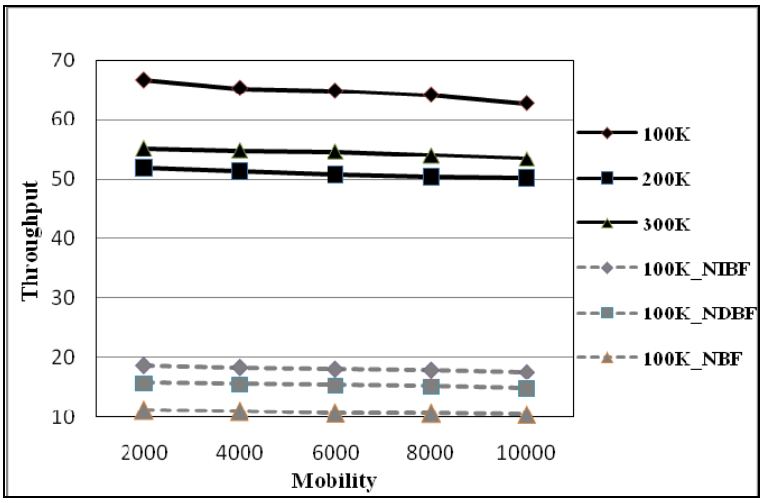
- **Data_size**: the number of initial moving objects and moving queries. It represents the size of the moving object database plus the number of continuous queries.
- **Q_size**: the side length of query window for each moving query. It simulates query ranges in different applications.
- **Mobility**: the total number of concurrent location updates for objects and queries in a batch. It corresponds to the frequency of object/query location updates.
- **OM_ratio**: the percentage of object location updates in Mobility. It reflects the relative update frequency between objects and queries.
- **QR_ratio**: the portion of query reports compared to Mobility. It shows how frequently the query report operation is requested.

Table 6. Experiment Parameters.

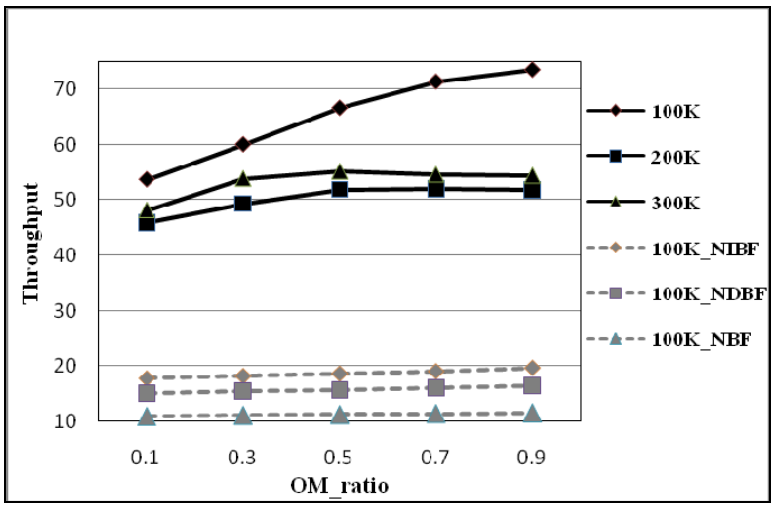
	Data_Size	Q_size	Mobility	OM_ratio	QR_ratio
Default	N/A	5	2K	50%	5%
Range	100K~300K	5~25	2K~10K	10%~90%	5%~25%

The proposed framework was implemented using JDK 1.5, based on the R-tree code from [5]. The experiment system was built on a desktop with a Pentium-D 2.8 GHz CPU and 2 GB main memory. Three sets of initial moving objects and moving queries were used in each set of experiments, with data_size 300K, 200K, and 100K, respectively.

4.4.1 Throughput vs. Buffers



a) Benefits of Buffers over Mobility



b) Benefits of Buffers over OM_ratio

Figure 28. Throughput vs. Buffers.

Experiments were conducted to evaluate the effectiveness of the *I-buffer* and *D-buffer* in the framework. Location updates and continuous queries were processed on the original C^3, C^3 without *I-buffer* (denoted as NIBF), C^3 without *D-buffer* (denoted as NDBF), and C^3 without any buffer (denoted as NBF), respectively. Since the mobility and OM_ratio have significant impact on the system throughput, these two parameters were varied in the comparison to analyze the benefits of the buffers.

Figure 28 a) shows the throughput of C^3 on the three datasets and those of the simplified C^3 on the 100K dataset when the mobility increased from 2,000 to 10,000. The *x*-axis shows the mobility, and the *y*-axis indicates the throughput. Generally, deactivating any operation buffer significantly increased I/O operations for tree updates. Furthermore, in C^3, the increased updates on the R-trees caused additional locks and lengthened the lock durations. As shown in both figures, by deactivating the *I-buffer*s, the system throughput decreased by more than 70% for the 100K dataset. When the *D-buffer*s were deactivated, the system lost about 75% of the performance. When there was no operation buffer applied, the system throughput degraded more than 80% from the original C^3 for the 100K dataset. As observed from the above results, the *D-buffer*s promoted the system performance slightly more than the *I-buffer*s. This is because the insert operations with *I-buffer* need to traverse the R-tree, although only the higher levels for most of the time, and the *I-buffer*s close to the R-tree root may become bottlenecks. On the other hand, the delete operations with *D-buffer* do not require tree traversal in most cases.

The comparison of different versions of C^3 when the OM_ratio gradually increased is illustrated in Figure 28 b), where the *x*-axis represents the OM_ratio and the *y*-axis shows the throughput. Following the trend of the original C^3 on the 100K dataset, the simplified versions of C^3 increased along with the OM_ratio. In addition, the NIBF outperformed the NDBF, and the NBF always had the lowest throughput.

4.4.2 Throughput vs. Mobility

In this set of experiments, the mobility of the objects and queries varied from 2,000 to 10,000 updates per batch, while the OM_ratio, QR_ratio, and Q_size were set to their default values. The throughput of the framework was measured on three datasets with different sizes, from 150K objects with 150K queries to 50K objects with 50K queries. The throughput of TD and VD on the same datasets and movements was also collected. The experiment results are shown in Figure 29, where the *x*-axis represents the mobility and the *y*-axis shows the throughput. The throughput on all datasets decreased linearly when the mobility increased. These results are considered as reasonable because a higher mobility indicates more location update operations in the processing queue. And a longer queue leads to more waiting time for each operation.

The throughput on both the 200K and 300K datasets reduced about 5% when the mobility changed from 2,000 to 10,000. However, the decreasing rate on the 100K dataset was about 10% with the increasing of mobility. This suggests that mobility affects more on the smaller datasets than the larger ones. It is because concurrent operations on a smaller dataset may cause more conflicts, and consequently increase the average

waiting time. Interestingly, the 300K dataset performed about 8% better than the 200K dataset. This is because the same number of update operations causes less pending locks on a larger dataset. These results demonstrated the scalability of the proposed framework regarding to the data_size.

Compared to the throughput of C^3, TD on the three datasets performed 10-15 less operations every second, and VD performed 6-17 less operations. Specifically, the throughput on 300K dataset lost about 25% by applying TD, and lost 15% ~ 28% by applying VD. The throughput on the 100K and 200K dataset dropped about 15% ~ 20% for TD and 12% ~ 27% for VD. Among the three approaches, VD decreased its performance most significantly when mobility increased. The low throughput of VD was caused by the large number of redo operations during frequent updates, and these redo operations may cause additional conflictions consequently. On the other hand, the decreasing rates of TD's throughput was higher than those of C^3, because a frequently updated dataset benefits more from reduced lock durations.

As shown from the figure, the throughput of C^3 on the 300K dataset was even better than the 200K dataset. These results demonstrated the scalability of the R-tree and the advantages of the lazy group update technique. The lazy group update approach minimizes the cost of R-tree update operations, and the search operation is facilitated by the finer leaf nodes on the R-trees with larger datasets. These advantages compensated the increased storage and overlaps among the tree nodes of the 300K dataset.

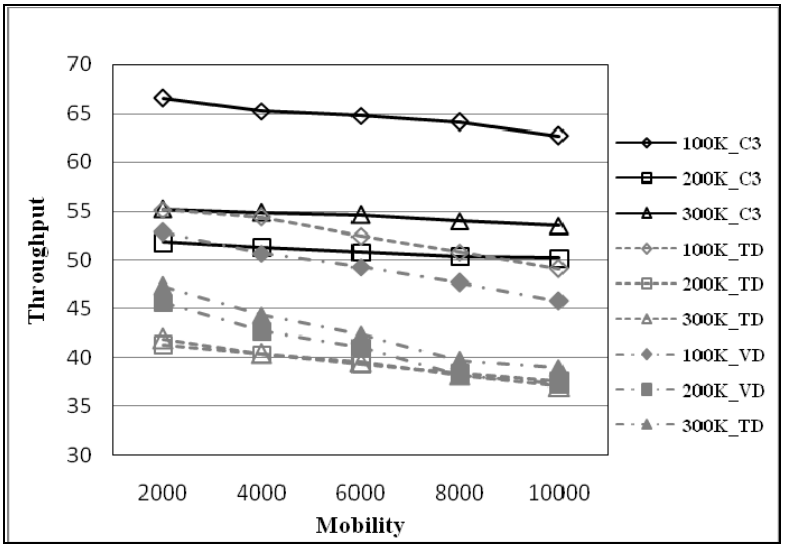


Figure 29. Throughput vs. Mobility.

4.4.3 Throughput vs. OM_ratio

In this set of experiments, three datasets with size 300K, 200K, and 100K, were used to evaluate how the OM_ratio affects the system throughput. Figure 30 illustrates the throughput of C^3, TD, and VD, with the OM_ratio varied from 10% to 90%. The *x*-axis indicates the OM_ratio, and the *y*-axis represents the throughput. As shown in the figure, when the portion of the object location updates in simultaneous operations increased, the throughput increased too. This is because an object location update usually costs less than a query location update. An object location update, based on the algorithm, performs a point insertion and a point query. On the other hand, a query location update inserts a window and processes a window query on the R-tree. Therefore, a query location update requires more I/O accesses and index locks.

Furthermore, from this figure, it is clear that the 100K dataset benefited more from the increase of the OM_ratio. The throughput of the 100K dataset was raised 40% by increasing the OM_ratio, while the 300K and 200K datasets only increased 15%. This difference was caused by the fact that the R-trees for the 100K dataset have less overlapped area among the tree nodes. Less overlapped area in the R-trees can cause fewer tree node accesses and locks during point operations. These results show that the performance of location management on a small dataset can be significantly improved by increasing the OM_ratio. Similar to the previous set of experiments, the throughput of the 300K dataset was better than the 200K dataset, with the same trend. Both these datasets exhibited consistent performance when the OM_ratio gradually increased from 50% to 90%, because the increased number of object location updates caused more conflicts with *X-lock*s on the *O-tree*, which compensated the benefit from fewer range query and update operations.

The throughput of TD and VD approaches in the figure shows the performance degrade from C^3. On all the three datasets, the throughput of TD was about 5-13 operations per second lower than C^3, and VD lost 2-11 per second from C^3. On the 100K dataset, the increase of the TD performance had the similar slope as that of the C^3, whereas the TD on the 200K and 300K datasets did not follow the trends of the corresponding C^3 performance. Different from C^3, the TD on the 200K and 300K datasets increased the throughput linearly along the OM_ratio, because TD always locks the accessed indices until commit and is less affected by the conflicts on different granules. The performance of VD was slightly worse than TD on the 100K datasets and better than TD on the 200K and 300K. This illustrated that VD had better scalability than TD, although performed worse on small datasets.

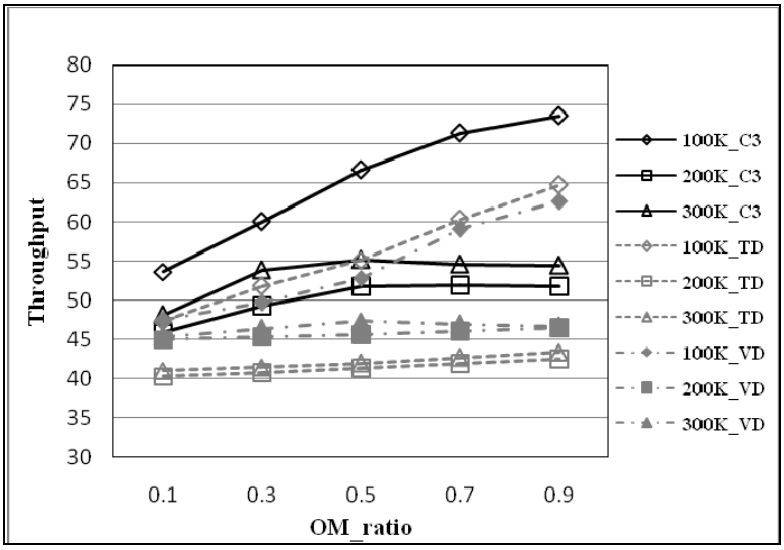


Figure 30. Throughput vs. OM_ratio.

4.4.4 Throughput vs. QR_ratio

This set of experiments examines the relationship between the QR_ratio and the system throughput. The throughput was measured while the QR_ratio increased from 5% to 25%. The results are illustrated in Figure 31, where the *x*-axis indicates the QR_ratio and the *y*-axis shows the throughput. Generally, a higher QR_ratio decreases the system performance, because more query reports are issued to consume the system resources. As shown in the figure, the throughput of C^3 on the three datasets decreased by 1-2 operations per second when the QR_ratio increased from 5% to 25%. These results suggest that the cost for query report operation is relatively low, so that it can be efficiently processed without significantly blemishing the system performance. This is the benefit from the design of this concurrent continuous query processing, because the *Q-result* always stores the correct results, and the query report operation only requests *S-lock*s on the corresponding *Q-result* entry and the *D-buffer* entries of the *O-tree*.

Similarly, although the number of query report operations in each batch increased from 100 to 500, there was no significant change on the throughput of TD and VD. As can be seen from the figure, C^3 improved the performance 16% ~ 26% from VD, and 22% ~ 33% from TD.

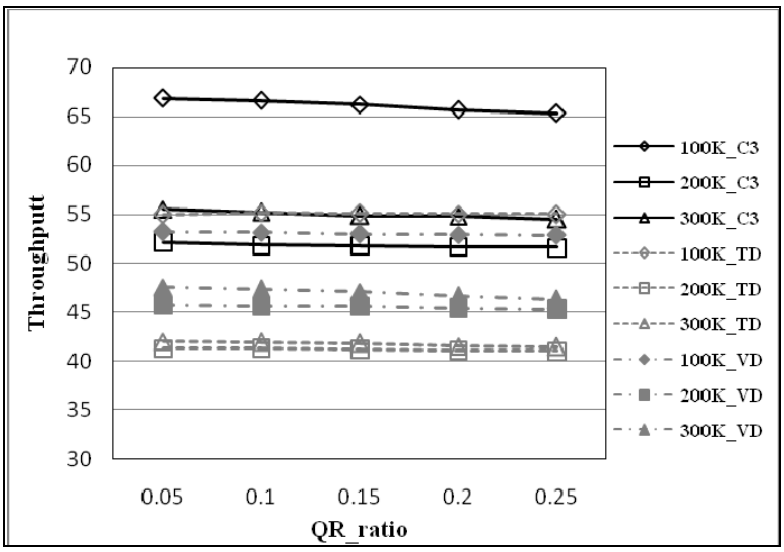


Figure 31. Throughput vs. QR_ratio.

4.4.5 Throughput vs. Q_size

This set of experiments varies the Q_size to study how the query window size affects the system performance. The experiment results are plotted in Figure 32, where the *x*-axis shows the Q_size and the *y*-axis represents the throughput. In this figure, the throughput of all the three datasets slightly and linearly decreased when the Q_size was increased from 5 to 25. The throughput of the 100K dataset reduced by one, and the 300K and the 200K datasets both reduced by less than 1. Once the Q_size increases, each continuous query may cover more objects, and each object movement may affect more queries. Therefore, not only the tree access cost, but also the number of requested locks will increase. However, the maximum Q_size 25, which matches the common continuous query applications, is still a small range query compared to the R-tree nodes. In this case, the impact of the Q_size on the system throughput is insignificant. Following the trend in the previous experiment results, the 300K dataset performed better than the 200K dataset under C^3, due to its fewer lock conflicts.

Similar to C^3, the performance of TD slightly degraded when the Q_size increased from 5 to 25. C^3 on each dataset achieved a significant performance improvement against TD. C^3 on the 100K dataset improved the throughput by 22% from TD, 24% on the 200K dataset, and 34% on the 300K dataset. Compared to VD, C^3 improved the performance by 26% on the 100K dataset, 16% ~ 19% on the 200K dataset, and about 20% on the 300K dataset.

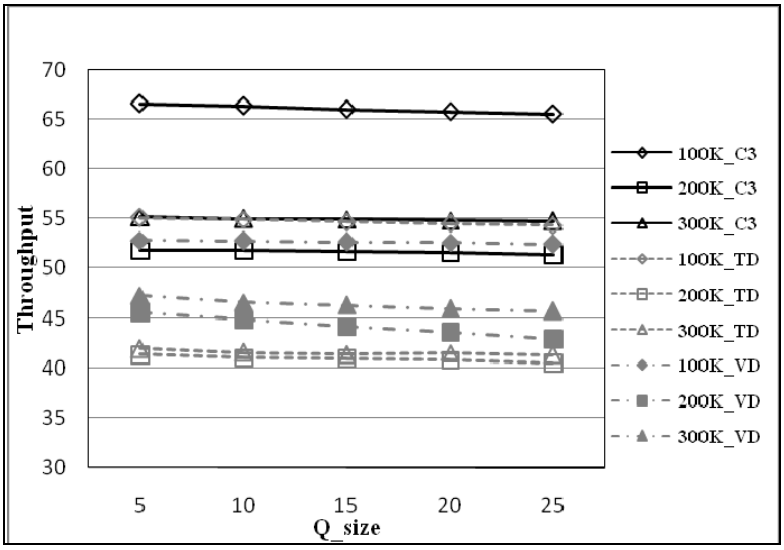


Figure 32. Throughput vs. Q_size.

The experiment results demonstrated that the performance of the proposed concurrent continuous query processing approach is efficient and scalable. As an interesting observation, in all these experiments, the 300K dataset outperformed the 200K dataset, which validated the scalability of the proposed approach in terms of data_size. Meanwhile, OM_ratio and mobility have noticeable impacts on the system throughput, while QR_ratio and Q_size do not significantly affect the performance. In addition, C^3 gains substantial benefits by applying optimal lock durations and utilizing the operation buffers in the framework.

Table 7. Impacts of Parameters

Parameters / Throughput	Data_size ↑	Mobility ↑	OM_ratio ↑	QR_ratio ↑	Q_size ↑
Change Trend	↓ or ↑	↓	↑	↓	↓
Change Speed	Fast	Fast	Fast	Slow	Slow

The experiment results demonstrated that the performance of the proposed concurrent continuous query processing approach is efficient and scalable. As an interesting observation, in all these experiments, the 300K dataset outperformed the 200K dataset, which validated the scalability of the proposed approach in terms of data_size. Meanwhile, as summarized in Table 7, OM_ratio and mobility have noticeable impact on the system throughput, while QR_ratio and Q_size do not significantly affect the performance. In addition, C^3 gains substantial benefits by applying optimal lock durations and utilizing the operation buffers in the framework.

4.5 Conclusion

We propose C^3, a concurrency control protocol for continuous queries, on an R-tree-based indexing framework. It is the first concurrency control protocol that protects the concurrent continuous query processing with lazy update techniques. It is proved to achieve serializable isolation, consistency, and deadlock-freedom for moving continuous queries over moving objects. Extensive experimental results on benchmark datasets have validated the efficiency and scalability of the proposed framework. This work provides an efficient solution for continuous query processing and promotes its applicability in multi-user systems.

Future efforts could focus on extending this protocol for other spatial operations such as continuous kNN search and spatial join to establish a complete concurrent access framework.

Chapter 5. INCREMENTAL kNN SEARCH ON LINEAR SPATIAL INDICES

An efficient kNN search algorithm is proposed together with several optimizations to enhance the applicability of B-trees in spatial databases. The optimization techniques for spatial operations on SFC, including multiple SFCs, query composition and bitmap, are proposed to reduce the I/O cost of tree traversals utilizing the traditional B+-tree/B^{link}-tree.

5.1 Preliminaries

5.1.1 Problem Definition

In order to propose appropriate solutions for query operations in a particular application, the kNN search problem and application environment must be specified. In this kNN problem, the data access method is based on SFCs and B+-tree. All the multidimensional data are mapped into one-dimensional space via SFCs, indexed by B+-tree, and stored on disk according to their SFC values. The execution of a kNN query will incrementally retrieve the k nearest neighbors of a given query point with as few I/O operations as possible. Formally specified, the input of this kNN search is a d-multidimensional point q, the number of nearest neighbors k, and a d-dimensional dataset S. The output of this kNN search consists of the k data points (o_1, o_2, …, o_k) nearest to q in S.

Three assumptions are made as follows as part of the detailed description of this problem. **First**, the multidimensional dataset contains points only, which means each data record is treated as a point in the multidimensional space. **Second**, space-filling curves are applied to provide the global order for B+-tree indexing (see Figure 33), and a standard B+-tree is used as the one-dimensional access method, as shown in Figure 34. The data space is divided into equal-sized boxes (cells); each of those is assigned a unique *ID* via an SFC. The order of SFC is determined by the density of the dataset, so that each cell will not contain more data points than can be stored in one data page. The leaf nodes of a B+-tree only contain the pointers to the data pages for non-empty cells. **Third**, an optimization is assumed to exist in the B+-tree access framework, which arranges the spatial data in disk pages as consecutive as possible during construction, and caches and sorts the physical addresses of data pages acquired from the B+-tree during retrieval, and then retrieves the actual data according to their physical locations. In this case, different SFC mappings can get the same I/O performance on data page retrieval for a set of cells as long as the data storage is given.

These assumptions are reasonable in real spatial applications, and they add constraints to the aspects of the problem.

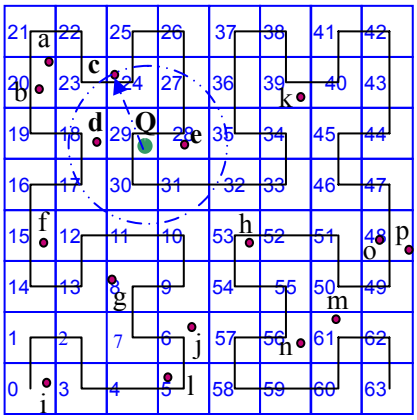


Figure 33. Examples of Point Datasets with Hilbert Curve Mapping.

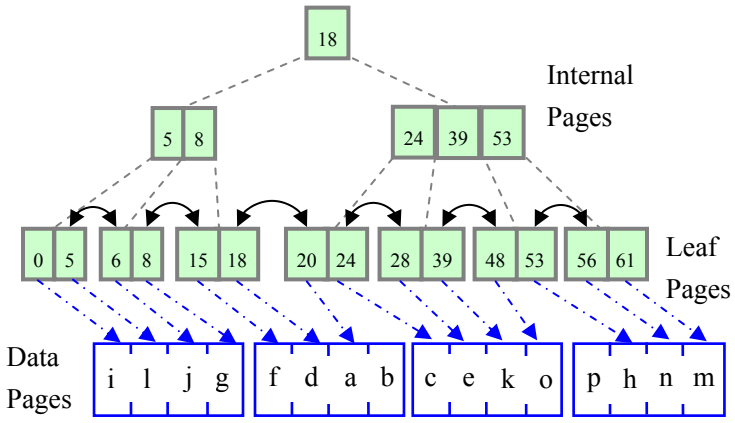


Figure 34. B+-tree for the Dataset in Figure 33.

5.1.2 SFC-Crawling Approach

Generally speaking, a kNN algorithm should determine a search space that is neither too small to miss correct answers nor too large to degrade performance. A straight-forward kNN search approach on spatial data indexed by SFCs was discussed in [68], namely, SFC-crawling. This approach utilizes the spatial locality provided by a Hilbert curve to retrieve objects in the range that is guaranteed to contain the *k* nearest neighbors. Specifically, given a query point, this approach will check the cells with adjacent Hilbert values to the query point in order to find the *k* nearest neighbors. The distance from the query point to the current *k*-th nearest neighbor then will be used as the radius of a circle. The MBR of this circle will be

searched to retrieve all the objects inside it. Finally, after sorting these objects according to their actual distance to the query point, the actual *k* nearest neighbors will be returned as the results. For example, in the data space in Figure 33, to find the two nearest neighbors of a query point *Q* located in the cell with Hilbert value 29, this approach first will scan cell 29 and then iteratively expand the scan along the curve until it finds 2 objects. In this case, the following sets of cells will be searched sequentially: {29}, {28, 30}, {27, 31}, {26, 32}, {25, 33}, {24, 34}, until the objects *e* (in cell 28) and *c* (in cell 24) are found. The search range will be calculated using the distance from *Q* to object *c* as the radius because the object *c* is the second nearest neighbor along the curve to *Q*. The cells covered by this circle then can be searched to retrieve all the objects inside. Finally, based on the distance, the objects *d* (in cell 18) and *e* (in cell 28) will be returned as the 2 nearest neighbors.

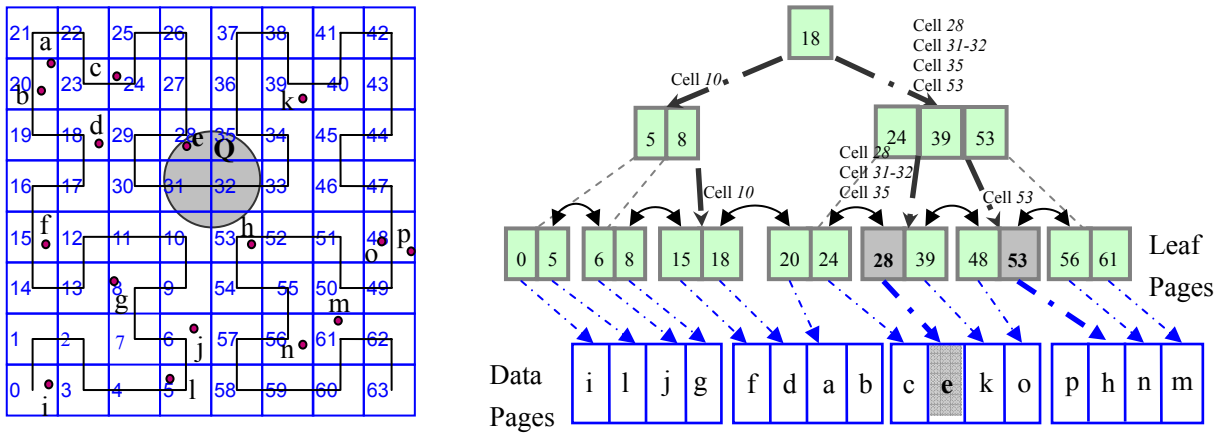
This SFC-crawling approach uses the Hilbert curve as a filter to estimate the search space and then validates the results with a real distance computation to ensure no real nearest neighbors are missed. However, although the spatial locality of a Hilbert curve can be used to reduce some unnecessary searching, it cannot guarantee an optimal search space as in some skewed distributions the *k* nearest neighbors along the curve may result in a much larger search space than necessary. Furthermore, SFCs other than Hilbert may further degrade the search performance because they may not preserve optimal locality.

5.1.3 Observation

Based on its scalability and locality preservation, Hilbert curves, with their zero Jump and high Still percentages [21], are suitable for most applications. However, according to the assumptions listed in the problem definition in Section 3.1, all the data objects are physically stored according to their indexed key, so a data object with an SFC value of *n* is allocated as immediately as possible before the data objects with value *n+1*, and after the data objects with value *n-1*. Meanwhile, an optimized schema accesses the actual data pages after caching and sorting the addresses. Therefore, for any SFC, the percentage of different types of segments may not affect the access performance on data pages under this circumstance. In this scenario, the I/O cost for data page retrieval is fixed by the set of non-empty cells accessed in a query.

In this kNN problem, as the cost of disk access to retrieve the result dataset is fixed by the set of visited non-empty cells, query performance is mainly determined by the number of pages accessed in the B+-trees. In terms of the general performance, Hilbert space-filling curves offer the optimal solution for range queries. However, as a recursive space-filling curve, Hilbert curves are unable to preserve the locality adequately around the border area among low-order sub-curves. Based on the definition of cluster in [20], as illustrated

in Figure 35 (a), query window Q (the gray circle) covers five clusters, cells 10, 28, 31-32, 35, and 53. Consequently, multiple B+-tree traversals (bold dashed lines and gray nodes in Figure 35 (b)), one along the left branch and four along the right branch, must be processed for this query. Furthermore, in an incremental kNN search, the spatial query ranges determined by the kNN query algorithm have special features, such as being strip-shaped, if they are not in the first query iteration, as will be shown in Section 5.3. Based on these features, SFCs other than Hilbert curves or combinations of multiple SFCs may achieve similar or even better query performance. This issue will be discussed further in the optimization of multiple SFCs in Section 5.3.



(a) Range Query on a Hilbert Curve (b) B+-tree and the Corresponding Traversals

Figure 35. Multiple Clusters on a Hilbert Curve and Tree Traversal in a Query Window.

On the other hand, the structure of the B+-tree provides the potential for further optimization. The links between sibling leaf nodes in a tree can be utilized to expedite range queries by expanding the traversal horizontally on the leaf level. Based on this feature, one large range query usually accesses fewer nodes in the B+-tree than a set of small range queries, even though both have the same coverage. Another feature of the B+-tree is that because of the lack of value scope in internal nodes, most queries cannot terminate their traversal until they reach the leaf level, even if they return empty sets. The ability to determine whether a query will miss or not would therefore reduce the I/O cost of tree traversal. Based on the above observations, we propose the use of the new, more efficient kNN algorithm presented in the following section.

5.2 Incremental kNN Search

5.2.1 Basic Incremental kNN Algorithm

As the exact results are required in this application, we propose an incremental kNN query processing algorithm based on a best-first-search followed by several optimization steps. To perform a best-first-search, a **priority queue** *PQ* is applied to buffer candidate neighbor objects and cells. It sorts these objects and cells based on their Euclidian distance to the query point. Each entry in *PQ* is an itemset-distance pair and takes the form (itemset *i*, distance *d*). In each iteration of the kNN search, the top entry in *PQ* will be popped up and a new entry generated by expanding the search range will be inserted into *PQ* if needed. If the corresponding items of the entry removed from *PQ* are data objects, they will be output as the nearest neighbors. Otherwise they must be cells, and the objects within these cells will be retrieved and inserted with their respective distances from the query point to *PQ*.

To guarantee that the top entry in *PQ* contains the nearest neighbor if the corresponding itemset contains data objects, a consistency requirement is applied to the distance calculation. This requirement specifies that the distance of a cell in *PQ* must be less than or equal to the distance of any data objects within that cell. In this way, if the distance of an object is less than that of a cell there cannot be any object in the cell that is closer to the query point than that object. Based on this requirement, the distance of a cell to the query point is calculated as the minimum distance between this cell and the cell containing the query point. The computational cost for distance calculation can be reduced by doing this, because the distances between the grid cells can be obtained from a simple function or a pre-calculated table. On the other hand, the distance of an object is calculated as the distance between that object and the query point. Therefore, given a query point *q* and an object *o* in this approach, the **consistency requirement** can be expressed as $dist(p, o) \geq dist(cell\_of\_p, cell\_of\_o)$, where the distance function *dist()* returns the minimum distance between two items in multidimensional space.

The incremental kNN search is processed as shown in Algorithm 9. Lines 1-3 **initiate** the priority queue with the cell containing the query point *q* and an empty result set for the consequent processing. The main part of the algorithm, lines 4-15 in Algorithm 9, iteratively pops out the priority queue *PQ* to generate nearest neighbors one by one, expanding *PQ* by adding more cells until either the *k* nearest neighbors have been identified or all the cells have been searched. Each iteration consists of two stages. The **first stage, top item checking,** pops out and checks the top entry in the current *PQ*, and query B+-tree if its itemset is a set

of cells, or returns it as a nearest neighbor otherwise (lines 5-11). In this stage, the type of the top entry in *PQ* will be checked to identify whether it is a set of cells or not. If it is a set of objects, based on the constraint requirement, they are the nearest neighbors in the unsearched dataset. Otherwise, it is a set of cells that have not been searched and have the shortest distances to the cell of *q*. These cells are then grouped into clusters based on their SFC values and each cluster is treated as one B+-tree query. The objects retrieved from these queries on the B+-tree will be added to *PQ*, as well as their distances to *q*. The **second stage, queue expanding,** finds the neighboring cells that have not yet been searched, and adds them to *PQ* if necessary (lines 12-15). In this stage, the cells adjacent to the searched area are identified. If their distances to the cell containing *q* are less than the distance between the current top entry in *PQ* and *q* they will be added to *PQ*, because they may contain objects that are nearest neighbors of *q*. After investigating the neighboring cells, the algorithm will re-initiate the iteration to generate the next nearest neighbor.

Algorithm IncrementalkNN (*q*, *k*, *T*)
Input: *q:* Query Point, *k:* Number of Nearest Neighbors, *T:* B+-tree,
Output: *S:* Set of k-nearest Neighbors.

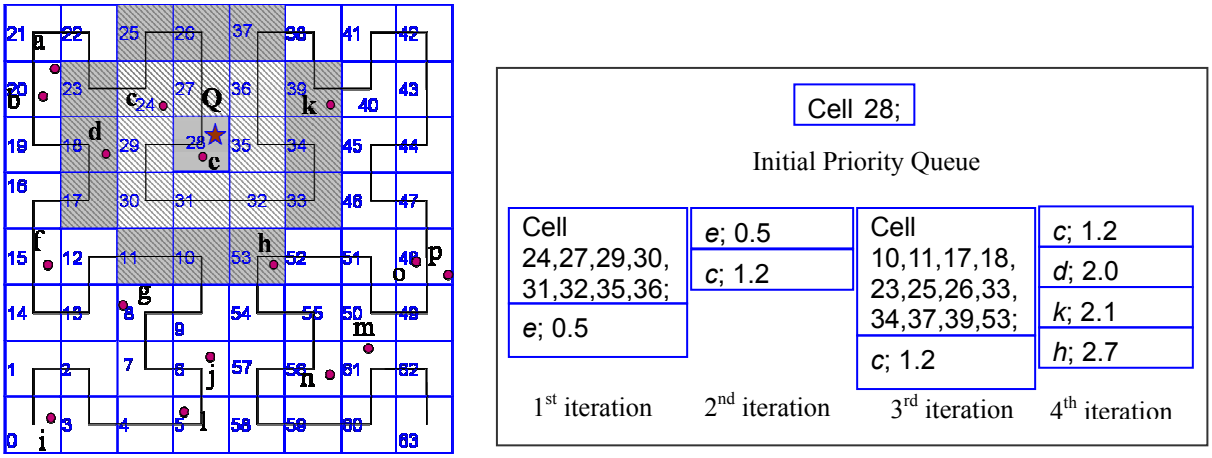
//Initiation
1. *PQ* = {(*cell_of_q*, -1)}; //initiate priority queue with the cell containing the query point
2. *S* = {}; //initiate result set
3. *CS* =set of all the cells in data space;

//Iteratively find nearest neighbors
4. While (|*S*| < *k* and (*CS* ≠{} or *PQ* ≠{}))
   //Stage1: top item checking in *PQ*
5. *e* = *dequeue(PQ)*; //get the nearest item
6. If (*e.itemset.type*="object")
7. *S* = *S* + *e.item*; //add the nearest object to *S*
8. else //*e.itemset.type* = "cells"
9. *OS*= *B+Query(e, T)*; //query objects in nearest cells
10. For each object *o* in *OS*
11. *PQ* = *enqueue(PQ, (o, dist(o, q)))*; //calculate exact distance of objects and put in priority queue
    //Stage2: queue expanding
12. *CQ* = {*c* | *c*∈*CS* and ∃ *dist(c, cell_of_q)*≤*g.dist*, *g*∈*PQ.top*}; //select neighboring cells
13. *CS* = *CS* - *CQ*;
14. For each cell *C* in *CQ*
15. *PQ* = *enqueue(PQ, (C, dist(C, cell_of_q)))*; //expand priority queue by adding neighboring cells
16. Return *S*;

Algorithm 9. Incremental kNN Search.

To demonstrate this process, an example of 2NN is given in Figure 36, where the star in cell 28 indicates the query point *Q*. Cell 28 will be put in *PQ* and assigned a distance -1 at initiation. Based on the algorithm, in the **first iteration**, cell 28 will be popped up and queried in the corresponding B+-tree in the function *B+Query()*. As shown in the figure, object *e* of this cell will be retrieved and added to *PQ*. Because the

distance between this object and *Q* is greater than 0, which is the distance from the shaded cells (cells 24, 27, 29, 30, 31, 32, 35, and 36) to cell 28 in Figure 36, these cells become the neighboring cells and are added to *PQ* with a distance 0. In the **second iteration**, these neighboring cells are popped up and queried via *B+Query()*. As illustrated in the figure, object *c* (in cell 24) will be retrieved and added to *PQ*. Because the distance of object *e* from cell 28 is less than 1, which is the distance of the striped and shaded cells in the figure, no more cells will be added to *PQ*. In the **third iteration**, the top of *PQ*, which is the object *e*, will be popped up and returned as the first nearest neighbor. Then the shaded and darkened cells in Figure 36 (cells 10, 11, 17, 18, 23, 25, 26, 33, 34, 37, 39, and 53) will be added to *PQ* with the distance 1 because the current top of *PQ* has a distance greater than 1. In the **fourth iteration**, these striped cells are popped up from *PQ* and queried via *B+Query()*. According to the data in Figure 36, objects *d* in cell 18, *k* in cell 39 and *h* in cell 53 will be retrieved and inserted into *PQ*. After this iteration, object *c* will be popped up and returned as the second nearest neighbor. Now there are *2* objects (from cells 24 and 28) in the result set and the algorithm terminates at this point.



(a) Expansion of Search Range (b) Priority Queue during Search

Figure 36. Example of Incremental 2NN Search.

The efficiency of a kNN search is mainly determined by the size of the space that the algorithm "overlooked." The ideal situation is that the searched area is a circle (or sphere when dimensionality>2) with the radius equal to the distance between the query point and the *k*-th nearest neighbor. However, the ideal situation is difficult to achieve. The cost of general best-first NN search has been analyzed in [69]. Different from that scenario, the hierarchical structure of the index tree applied in the proposed approach does not match the spatial locality, therefore, the complexity analysis based on the SFC space and this particular kNN search algorithm is provided as follows. For simplicity, a uniform distribution is assumed in the analysis. However, complexity for non-uniform distribution with known density function can be estimated in a similar way. The space complexity of this incremental kNN search algorithm is the size of

PQ, which is the maximum number of objects in the searched area. Assuming the spatial data points are uniformly distributed in 2D space with density *n*, and the side length of each square cell in the space is *l*, we can estimate the number of objects maintained in *PQ* when the radius of the current search range is *r1* and the radius of the last search range is *r2*. In the proposed algorithm, the objects maintained in *PQ* will be the union of the objects covered by the newly scanned cells and the objects with distance greater than *r1* that have been scanned. The number of newly scanned cells *nc* can be estimated as the number of cells intersected by the border of the current search range, minus the number of cells intersected by the border of the last search range. As discussed in [70], the number of cells intersected by a circle with radius *r1* is equal to $\left\lfloor \frac{2*r1}{l} \right\rfloor + 2$. Therefore, the number of newly scanned cells *nc* = $\left\lfloor \frac{2*r1}{l} \right\rfloor - \left\lfloor \frac{2*r2}{l} \right\rfloor$, and the number of new objects $newObj = n*l*l*nc$. The second part in *PQ* represents the objects that have been scanned but have distances greater than *r2*. This part can be estimated as the number of objects with distance less than *r1* and greater than *r2*, and these objects are not covered in the newly scanned cells. As it is safe to assume that half the objects in the newly scanned cells will be located within the current search radius *r1*, the number of objects in this part can be calculated as $oldObj = n*\pi*(r1^2-r2^2) - 0.5*newObj$. Finally, the size of *PQ* can be estimated as *newObj*+ *oldObj*, which equals $0.5*n*l*l*nc + n*\pi*(r1^2-r2^2)$, when $r1 >> l$. From this expression, it is clear that the space complexity of this approach is $O(n*l)$ and reducing the pace (i.e., (*r1-r2*)) in each iteration will lead to a smaller *PQ*. The space complexity of a non-uniformly distributed dataset can be estimated in a similar manner by replacing the density with an appropriate density function.

The upper bound of the processing time is determined by the number of cells that have been scanned, because each cell is likely to require at most one tree traversal, assuming the I/O cost is much higher than the computation cost [69]. Given the number of nearest neighbors *k*, the radius of the last search can be estimated as $r = \sqrt{\frac{k}{n*\pi}}$. The cells that have been scanned are expected to cover the area in this circle, as well as half of the cells intersected by the border of this circle. Therefore, the number of cells that have been scanned can be calculated as $\frac{\pi*r^2}{l^2} + 0.5*(\left\lfloor \frac{2*r}{l} \right\rfloor + 2)$, which indicates that the time complexity is $O(\frac{k}{n*l^2})$. This algorithm applies the best search approach to minimize both the search pace and search area, and consequently optimizes the processing time and space consumption. The effect of the dimensionality of the dataset depends on the SFC mapping function, which has been extensively discussed in the related literature [20, 21, 53].

5.2.2 Correctness

The correctness of this incremental kNN search algorithm is assured by the design of the priority queue and the consistency requirement. The priority queue, which is sorted using the distance function of the items, guarantees that only the items with the minimum distance will be retrieved or returned. The consistency requirement of the distance function guarantees that the object with the smallest value of the distance function will be identified as the nearest neighbor. The correctness can be expressed through the following lemma.

Lemma 1. In the proposed algorithm, if the top itemset in the priority queue for query point *q* includes object *A*, *A* must be the current nearest neighbor of *q*.

Proof:

∵ Itemsets in the priority queue are sorted by the outputs of the distance function.

∴ If the top itemset in the priority queue contains object A, for each object o in the priority queue, $dist(A, q) \leq dist(o, q)$; for each cell c in the priority queue, $dist(A, q) \leq dist(c, cell\_of\_q)$.

∴ Based on the definition of function *dist()*, ***A* is nearer to *q* than any object in the priority queue**.

∵ Based on the consistency requirement, for any object o_c in cell c, $dist(o_c, q) \geq dist(c, cell\_of\_q)$

∵ $dist(o_c, q) \geq dist(A, q)$

∴ ***A* is nearer to *q* than any object in the cells contained in the priority queue**.

∵ Based on the expanding strategy of the priority queue, for any cell c_i in the priority queue and any cell c_o outside the priority queue, $dist(c_o, cell\_of\_q) \geq dist(c_i, cell\_of\_q)$

∴ ***A* is nearer to *q* than any object in the cells outside the priority queue**.

□ **Q.E.D.**

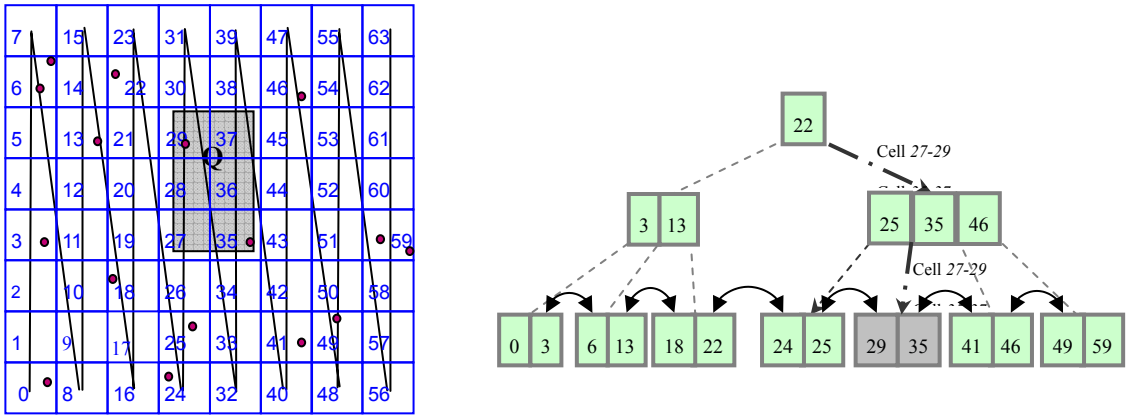
Since the priority queue contains all the un-returned objects or their cells in the searched area, if there is a current nearest object to q, it or its cell must appear in the priority queue. After consequently processing the kNN search, this nearest neighbor will appear at the top of the priority queue, because 1) if this object is in the queue, after expanding all the cells with the smaller distances to q, it will become the item with the smallest distance to q in the queue; 2) if its cell is in the queue, after expanding this cell, the status becomes the previous situation. Furthermore, since there are no other objects with smaller distance than this nearest neighbor to q, nothing will be returned before this nearest neighbor is returned. Therefore, the proposed

kNN search algorithm is assured to return the valid nearest neighbors to q in the dataset.

5.3 Optimizations

Several optimization techniques, namely multiple SFCs, query composition, and bitmap, can be used to reduce the I/O cost of spatial query processing on SFCs and B+-tree. These optimization steps compensate for the loss of locality from SFC mapping and utilize the features of the B+-tree.

5.3.1 Multiple SFCs



(a) Range Query on Sweep Curve (b) Corresponding B+-tree

Figure 37. Sweep Curve for Query and Data in Figure 33.

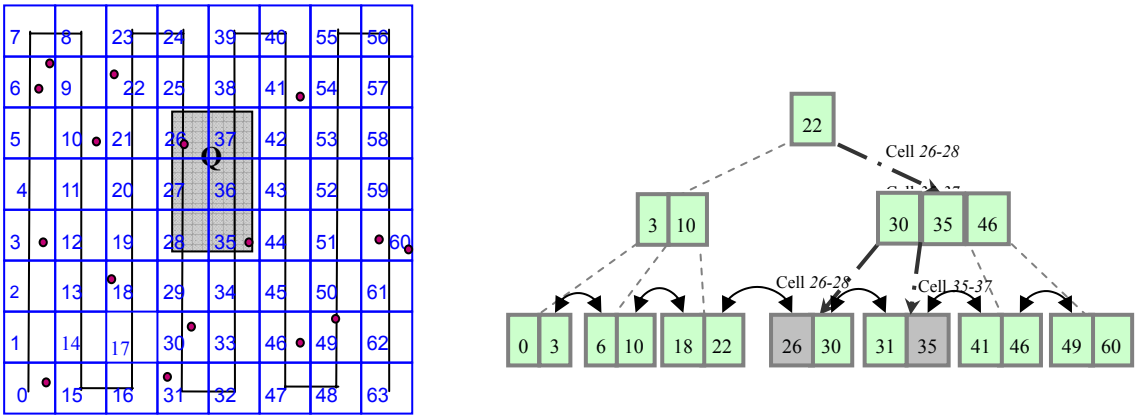
One significant issue in applying linear mapping to index multidimensional data is that multiple clusters may be accessed in a single spatial query. As multiple clusters will be considered as multiple queries on the B+-tree, reducing the number of clusters in a query could improve the query performance in some cases. As one solution, multiple space-filling curves can be applied to the same data space. Each SFC has an independent B+-tree with which to index the data objects based on its corresponding curve. Given a kNN query, the number of clusters in this range on different SFC curves can be evaluated. The curve with fewer clusters then can be chosen to map the query, and the corresponding B+-tree will be selected to execute the query. In an incremental kNN search, the appropriate SFC is selected every time when the search range is expanded. Observing the Hilbert curve, it clearly preserves the locality better in a low-order sub-curve (e.g., cells 0-15 in Figure 33) than between sub-curves (e.g., the query window in Figure 35). Therefore, as an auxiliary space-filling curve to the Hilbert curve, the curve applied should reduce the number of clusters around the connection area among low-order Hilbert sub-curves. Using non-recursive curves, for example Sweep (Figure 37) and Scan (Figure 38), there will be only two clusters (cells 26-28 and cells 35-37) in the

query range Q in the figures, fewer than the five clusters in the corresponding Hilbert curve (shown in Figure 35). Because a Scan curve preserves the locality around the border of a data space better than a Sweep curve (based on the curves in Figure 37 and Figure 38), it is therefore appropriate to choose a Scan curve as the auxiliary linear mapping function. This optimization needs one particular B+-tree for each SFC mapping function, which increases the construction and maintenance cost. However, it does not increase the query cost because every spatial query only accesses the B+-tree that fits the query best, and according to the assumptions in Section 3, the cost to access data pages in each query will be the same for different B+-trees. Furthermore, using multiple SFC could further enhance the query performance in multi-processor systems by parallelizing operations/suboperations on different B+-trees.

Procedure B+Query_MultiSFCs *(e, T, T')*
Input: *e:* Set of Cells, *T:* B+-tree on SFC_1, *T':* B+-tree on SFC_2,
Output: *S:* Set of Objects Contained in *e*.

1. $S = \{\}$;
2. Sol_1 = sets of consecutive cells in *e* according to SFC_1;
3. Sol_2 = sets of consecutive cells in *e* according to SFC_2;
4. If ($Sol_1.length < Sol_2.length$)
5. For each set of cells *CS* in Sol_1
6. $S = S + B+\_Search(CS, T)$;
7. Else
8. For each set of cells *CS* in Sol_2
9. $S = S + B+\_Search(CS, T')$;
10. Return *S*;

Algorithm 10. Procedure B+Query_MultiSFCs.



(a) Range Query on Scan Curve (b) Corresponding B+-tree

Figure 38. Scan Curve and Corresponding B+-tree for the Query and Data in Figure 33.

To apply this optimization, the procedure *s*hown in Algorithm 10 is required to replace the *B+Query(e,T)* in

the original kNN search algorithm (Line 9 in Algorithm 9). There is no limit to the number of SFCs that can be applied. However, here two SFCs are used to illustrate the concept. Both of the corresponding B+-trees are input to the optimization. Initially, the number of clusters in the cell set is calculated based on both SFCs (Lines 2-3 in Algorithm 10). Then the solution with fewer clusters is selected, and the corresponding B+-tree is traversed to retrieve the objects (Lines 4-9).

5.3.2 Query Composition

In order to reduce the number of B+-tree queries in one spatial query, another optimization approach is to merge discontinuous but nearby clusters into a single large range. Because the B+-tree has pointers to link sibling nodes from left to right at the leaf level, a one-dimensional range query only needs to traverse B+-tree once, then follow the links on the leaf nodes to access other leaf nodes until it reaches the end of the range. Utilizing this feature, the cost of tree traversal in queries of multiple clusters can be reduced by the need to query only one large cluster. For example, in Figure 35 it is better to query cells (28-35), which traverses the B+-tree once, rather than query cells (28, 31-32, 35), which traverses the B+-tree three times. For incremental kNN queries, after the queries are combined all the objects returned from the B+-tree are added to the priority queue without filtering, because the objects from cells (29, 30, 33, and 34), which are not in the original ranges, are likely to be queried when the search range is expanded in the next iteration. Having these objects in the priority queue may avoid the need for future B+-tree queries by bypassing these cells. For range queries, a filter step will be needed after executing this merged query in order to ensure that only the objects in the original range are accessed. To determine whether to merge two clusters or not, the I/O costs of the two strategies should be estimated. In order to make a rational choice, a lemma for cost estimation is therefore given as follows.

Lemma 2: The cost of B+-tree traversal utilizing sibling links between leaf nodes for range query (*h, h+I*) is

$$Number\_of\_Node\_Access(h, h+I) = L + I * NEC / (C * TC),$$

where *L* is the height of the B+-tree, *NEC* is the number of non-empty cells in the entire space, *C* is the average capacity of each leaf node, and *TC* is the total number of cells in the space.

Proof:

∵ The average number of consecutive cells that will contain one non-empty cell is TC / NEC, and C is the average number of cells in each leaf node.

∴ The scope of SFC values within one leaf node is $C * (TC / NEC)$.

∴ Given two cells with SFC values v_1 and v_2, where v_1-v_2 =I, the average number of leaf nodes between these two cells is $I / (C * (TC / NEC)) = I * NEC / (C * TC)$.

∵ Cost of tree traversal contains two parts, that incurred by moving from the root to a leaf node that contains h, and that incurred by moving left-to-right from the leaf node that contains h to the leaf node that contains $h+I$.

∵ The number of nodes accessed in traversal from the root to a leaf node is the height of the tree L.

∴ *Number_of_Node_Access(h, h+l),* the overall number of nodes to access, *is* $L+ I * NEC / (C * TC)$.

□ **Q.E.D.**

Procedure B+Query_QComp *(e, T)*
Input: *e:* Set of Cells, *T:* B+-tree,
Output: *S:* Set of Objects Contained in *e*.

```
1.  S = {};
2.  Thres = L * C * TC / NEC; //calculated from T
3.  e = sorted cells in e by SFC values;
4.  Clu = c0; //put the first cell of e into Clu
5.  For each cell ci in e, where i > 0
6.     If (ci-1 - ci > Thres)
7.        S = S + B+_Search(Clu, T);
8.        If (i == e.length)
9.           S = S + B+_Search(ci, T);
10.    Else
11.       Clu = Clu + ci;
12.       If (i == e.length)
13.          S = S + B+_Search(Clu, T);
14. Return S;
```

Algorithm 11. Procedure B+Query_QComp.

Because the cost of tree traversal for two separated clusters is $2*L$, for the case where each cluster is covered in one leaf node the decision can be made by comparing 2 * L and L + $I * NEC / (C * TC)$. Thus, when I is less than $L * C * TC / NEC$, it would be advantageous to merge these two clusters. In queries with more than two clusters, this estimated cost comparison can be performed iteratively. As these parameters should be available in the metadata of the indexing tree, L and $NEC / (C * TC)$ also can be determined after the construction of the B+-tree. For example, for the dataset and index shown in Figure 33, L is 3, C is 2, TC is 64, and NEC is 14, therefore the threshold of I can be calculated as 27.4. This approach is expected to perform better in uniformly distributed data space, where the estimated cost will be close to the actual cost.

Additional CPU cycles are expected to filter out the retrieved objects not in the original search ranges. However, the I/O cost, which is usually much higher than the computational cost, is regarded as the dominant measure for the query composition.

The procedure *B+Query_QComp(e,T)* shown in Algorithm 11 is used to add the query composition optimization to the original kNN search by replacing the *B+Query(e,T)* (Line 9 in Algorithm 9). In this procedure, the threshold for compositing discussed above is calculated in Line 2. Gaps between adjacent cells in the sorted cell list are then checked to see whether they exceed the threshold. If a gap between two consecutive cells is larger than the threshold, the cluster that contains the first cell will be grouped into a single range query (Lines 7-9). Otherwise, the second cell will join the cluster that contains the first cell (Lines 11-13).

5.3.3 Bitmap

Building a bitmap for the cell utilization is another optimization that can be used for cases where the dataset has a skewed distribution over the whole space. In a skewed data distribution, the empty cells within the search range will still be queried in the B+-tree and identified as empty until the traversal reaches the leaf nodes in the tree. Consequently, these empty cells increase the number of unnecessary accesses during tree traversal. Because there is no exact range of values maintained in the internal nodes of the B+-tree, in some worst cases the tree must access the leaf node before discovering the key does not exist. Maintaining a bitmap to record the empty cells is expected to reduce this cost. Once a query is received and the query range is transferred into curve values, the system can check the utilization bitmap to filter the curve values of empty cells and send only the remaining values to the B+-tree.

0	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0
0	1	0	1	0	0	0	0
0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
0	0	1	0	0	0	0	0
0	0	0	1	0	1	1	0
1	0	0	1	0	0	0	0

Figure 39. Bitmap for the Data in Figure 33.

Building the bitmap requires either scanning the whole dataset or incrementally updating it along with the indexing of the dataset. Physically, a bitmap is an array of bits, where each bit denotes whether the corresponding cell is empty or not. The dimensions of this array equal the dimensions of the data space, so the size of the bitmap will be relatively small compared to the mainstream hardware configurations. For example, if a two-dimensional grid space has $2^{10} * 2^{10}$ cells, the size of the corresponding bitmap will be 128KB. Figure 39 gives an example of a bitmap constructed for the sample data in Figure 33. All the cells that contain data objects are marked as 1, with 0 denoting an empty cell. When this bitmap is checked before the range queries shown in Figure 35 and Figure 38, only two cells should be queried in B+-trees in both cases (cells 28 and 53 for a Hilbert curve, and cells 26 and 35 for a Scan curve), rather than the several range queries described in the previous sections. Practically, the bitmap can be implemented using a compression technique [71] to reduce the memory consumption and the effect of dimensionality.

This optimization is formally described in Algorithm 12, which is applied in the original kNN search algorithm by replacing the *B+Query(e,T)* (Line 9 in Algorithm 9). To invoke this procedure, a bitmap *Bm* generated outside of this procedure by scanning the dataset to identify the non-empty cells must be provided. Before grouping the cells into clusters, the empty cells are removed from the cell set *e*. Therefore, each traversal on the B+-tree is guaranteed to retrieve one or more objects.

Procedure B+Query_BM *(e, T, Bm)*
Input: *e:* Set of Cells, *T:* B+-tree, *Bm:* Bitmap,
Output: *S:* Set of Objects Contained in *e*.

1. *S* = {};
2. *e* = *e* – *Bm.empty_cells*; //remove empty cells
3. *Sol* = sets of consecutive cells in *e* according to SFC;
4. For each set of cells *CS* in *Sol*
5. *S* = *S* + *B+_Search(CS, T)*;
6. Return *S*;

Algorithm 12. Procedure B+Query_BM.

The above three optimizations are suitable for both range queries and kNN queries because they focus on ways to reduce the B+-tree traversals based on the features of SFCs. Multiple SFCs benefit from a general optimization that is independent of the data distribution. Query composition estimates the costs of tree traversals using the average data density and so performs better in uniformly distributed datasets, whereas bitmaps, which utilize the data distribution to reduce B+-tree traversals, are more suitable for sparse datasets. Experiments conducted to assess these methods are presented in the next section.

5.4 Experiments

To evaluate the performance of the proposed incremental nearest neighbor search and the effectiveness of the optimizations, a series of experiments on real datasets were conducted by comparing the I/O costs among different kNN queries, as well as different optimizations, as shown in Figure 40. Additionally, comparison against the existing kNN search on R-tree showed the applicability of the proposed approach.

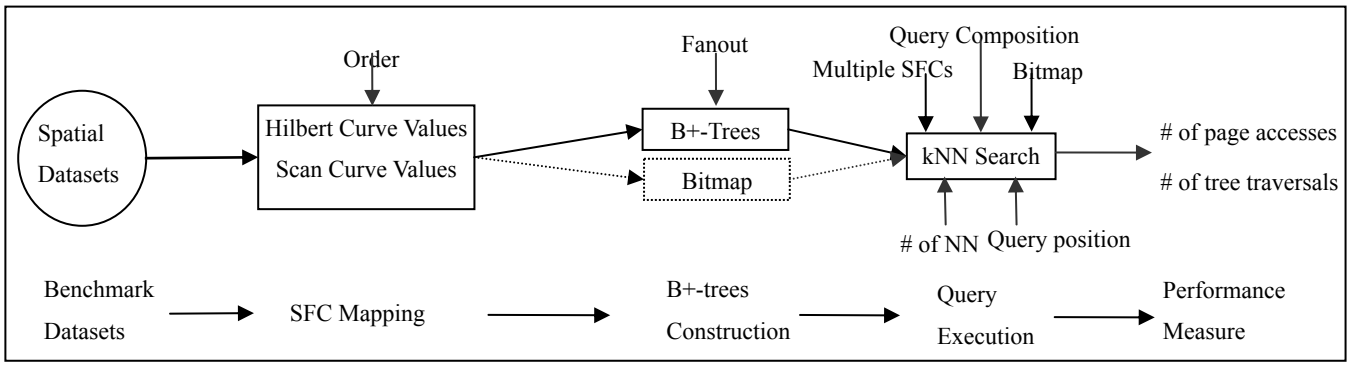


Figure 40. Experiment Flow.

Datasets: The two real datasets used in the experiments are the City of Oldenburg, Germany, containing about 6,000 road network nodes, and California Places, containing more than 62,000 points of interest in California [22] (Figure 41). The data points for the road network in the City of Oldenburg are relatively uniformly distributed, with about 40% of the cells being empty. In contrast, the dataset for points of interest in California has a skewed distribution, with about 80% of the cells being empty.

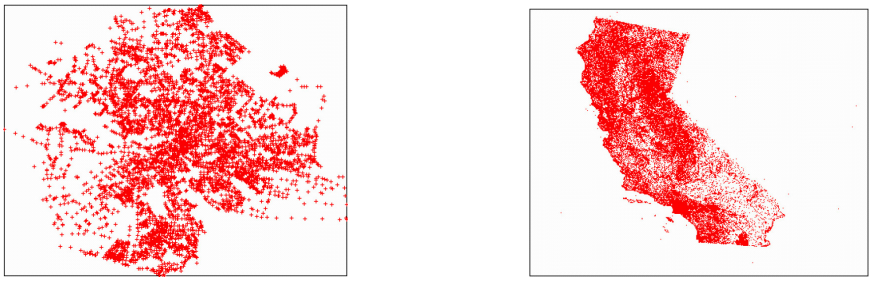
SFC Mapping: Both datasets were mapped using both Hilbert and Scan curves. The orders of the SFCs were defined based on the density of the datasets. Taking into account the appropriate average number of points in each cell, the City of Oldenburg dataset was mapped in SCF with order 5, while the California Places dataset was mapped with order 8.

B+-tree Construction: Based on the SFCs, B+-trees with a height of 3 were built on the City of Oldenburg dataset, and trees with a height of 4 were built on the California dataset. The fanout of the B+-trees built in the experiments was 32, and the size of a disk page was set as 1,024 Bytes. The fanout was determined such that a reasonable tree height could be maintained based on the datasets.

Query Execution: In the experiments, the testing kNN queries were generated by assigning different numbers of nearest neighbors (k) and randomly selecting query points in the datasets. The number of nearest neighbors was varied from 20 to 500 in order to comprehensively assess the performance of the different approaches. The query points for each dataset were randomly selected from the center points of non-empty cells in the data space.

Performance Measurements: The number of tree traversals and number of page accesses in B+-trees are commonly used to measure the I/O cost because the tree traversal is the only I/O consuming part in incremental kNN queries except for the access to physical data held in storage, which is fixed for a particular result set. It is known that the use of a disk page buffer may improve the query efficiency by caching visited paths on the B+-tree, but this was not done here in order to evaluate the corresponding effectiveness of the proposed optimizations.

The experiments were conducted on a Pentium 4 desktop with 512MB memory, running a Java2 platform under Windows XP™. The implementation of the B+-tree was based on a plain file system.



(a) City of Oldenburg (b) California Places

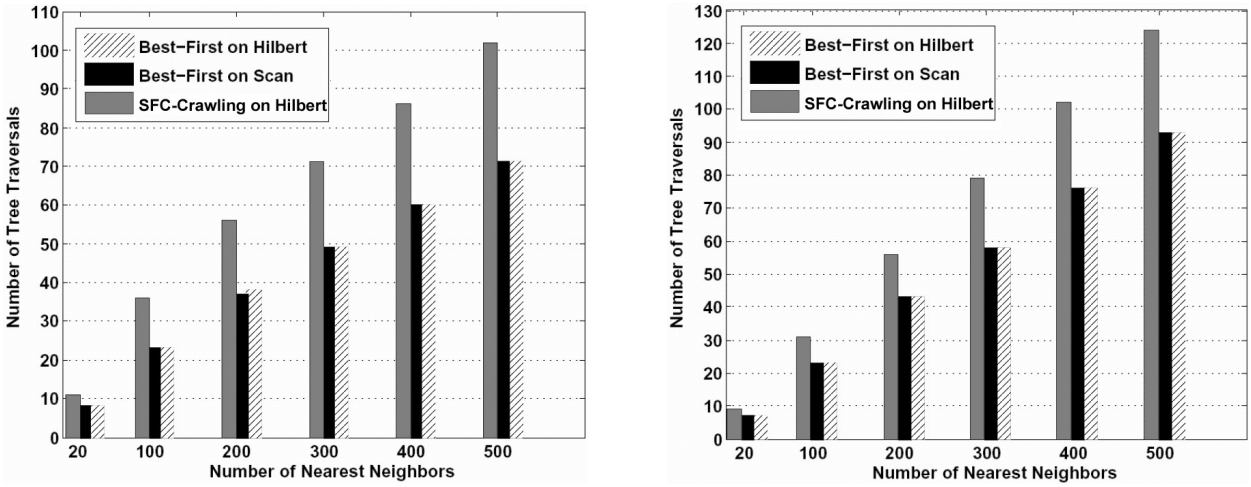
Figure 41. Datasets.

Two sets of experiments are described in the following subsections. The first set of experiments illustrated the performance and scalability of a proposed incremental kNN query on both Hilbert and Scan curves by examining their numbers of tree traversals and page accesses. The second set of experiments demonstrated the effectiveness of the three optimizations, as well as their combinations, by comparing the numbers of page accesses.

5.4.1 Scalability

In this set of experiments, both Hilbert and Scan curves were applied to map the datasets, and B+-trees were constructed accordingly. The proposed best-first kNN search based on different SFCs was then compared against the SFC-crawling approach (introduced in Section 5.1.2). In order to evaluate their

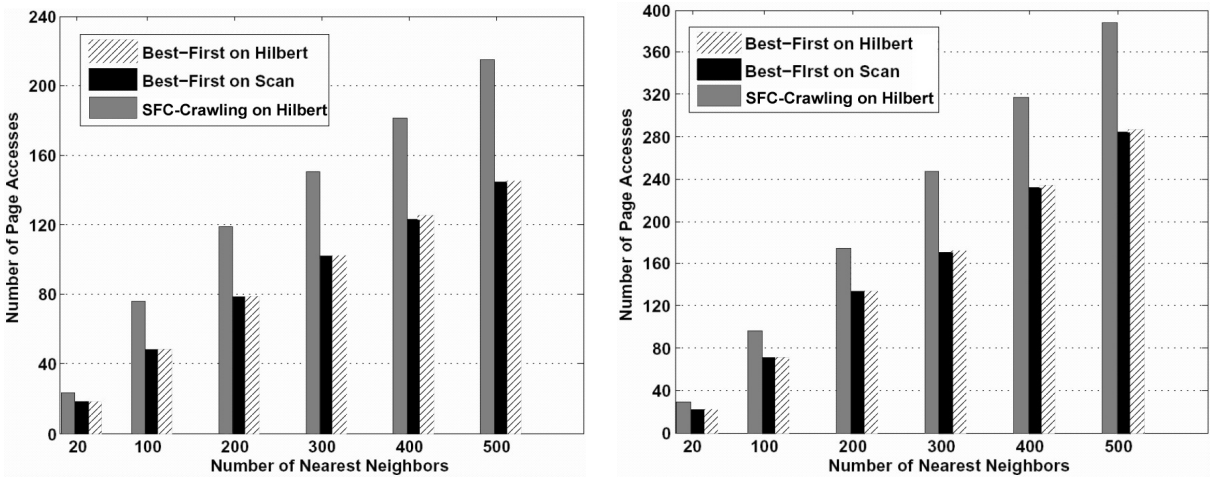
performance, the number of tree traversals and page accesses were recorded, while the numbers of nearest neighbors required varied. The numbers of tree traversals for both the proposed approach and the SFC-crawling approach are plotted in Figure 42, where the X-axis represents the number of nearest neighbors in the query and the Y-axis shows the number of tree traversals. The numbers of page accesses for these approaches are presented in Figure 43, where the X-axis denotes the number of nearest neighbors and the Y-axis indicates the number of page accesses. In these figures, the I/O costs, in terms of the number of tree traversals and the number of page accesses, of the proposed kNN search on both Hilbert and Scan curves increased linearly with the growth of the number of nearest neighbors. These results show that in datasets with different sizes and distributions, the proposed incremental kNN search has linear scalability, which is primarily benefited from the way it expands the search range and utilizes the priority queue. However, even though the SFC-crawling approach also showed linear scalability, it exhibited 40-50% more I/O cost than the incremental kNN search in most cases. This was because the SFC-crawling approach cannot find an accurate approximation of the search range in most cases.



(a) # of Tree Traversals on City of Oldenburg Data. (b) # of Tree Traversals on California Data.

Figure 42. Numbers of Tree Traversals with Hilbert Curve, Scan Curve and the SFC-crawling Method.

As discussed in Section 5.1, the Hilbert curve may not be the only suitable choice in incremental kNN queries, and these experimental results provided support for this statement. On both datasets, as can be observed from these figures, an incremental kNN search on Scan curves generally achieved the same performance as on Hilbert curves, and was even slightly better in several cases (such as on 200 nearest neighbors in Figure 42 (a) and on 400 nearest neighbors in Figure 43(a)). Based on these experimental results, therefore, Scan curve offers an effective alternative to the Hilbert curve in two-dimensional incremental kNN searches, considering the complex calculations involved in the Hilbert curve.



(a) # of Page Accesses on City of Oldenburg Data. (b) # of Page Accesses on California Data.

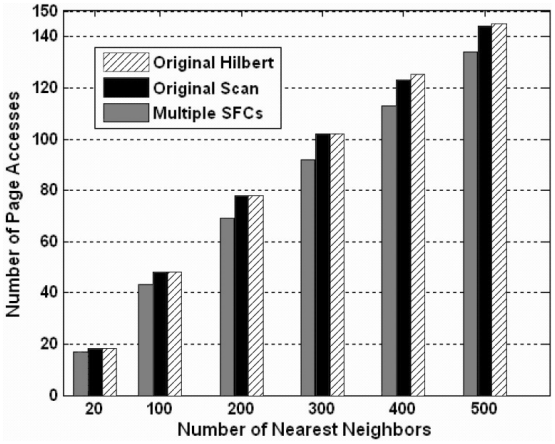
Figure 43. Numbers of Page Accesses with Hilbert Curve, Scan Curve and the SFC-crawling Method.

5.4.2 Effectiveness of Optimizations

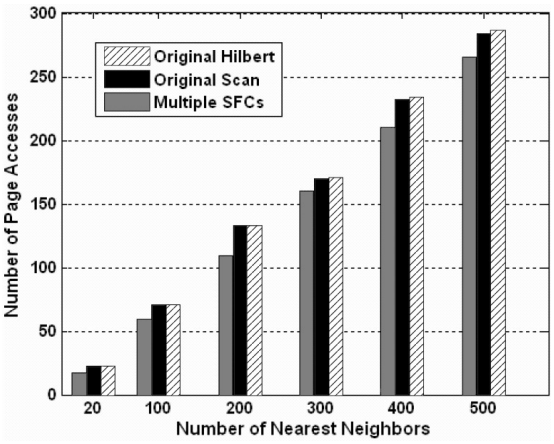
In this set of experiments, the three optimizations designed in Section 5.3, namely multiple SFCs, query composition, and bitmap, were applied on both datasets to determine their effectiveness.

5.4.2.1 Multiple SFCs

The optimization with multiple SFCs was applied on the original incremental kNN search approach. Hilbert curves and Scan curves were used as multiple choices in the experiments, and the set of B+-tree queries in each kNN search iteration selected the one with fewer clusters to query the corresponding B+-tree. The numbers of page accesses for kNN queries with multiple SFCs are compared with the original approach on both Hilbert and Scan curves in Figure 44. As illustrated in the figure, the kNN search with multiple SFCs generally achieved better performance than the original method. Specifically, in most cases this optimization had about 10% fewer page accesses compared to the original method. The use of multiple SFCs could achieve a better performance if, within each search iteration, several SFCs could participate to share the range queries without competing to exclusively execute the queries. However, this would require complex arrangements to be made in order to generate an optimal plan for assigning cells to different SFCs. The experimental results in this figure show that this coarse-version optimization with multiple SFCs can effectively improve query performance on different datasets.



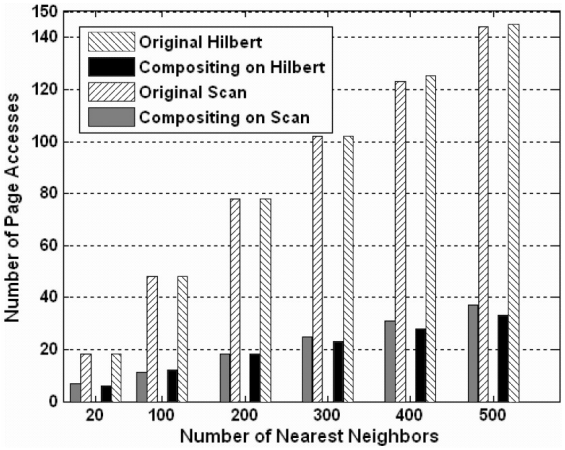
(a) # of Page Accesses on City of Oldenburg Data.



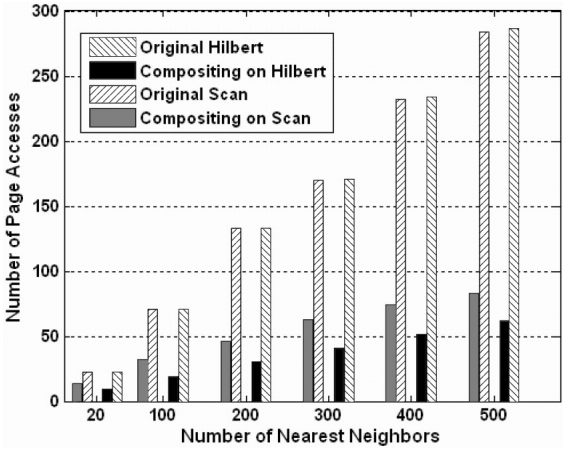
(b) # of Page Accesses on California Data.

Figure 44. Numbers of Page Accesses with Multiple SFCs.

5.4.2.2 Query Composition



(a) # of Page Accesses on City of Oldenburg Data.



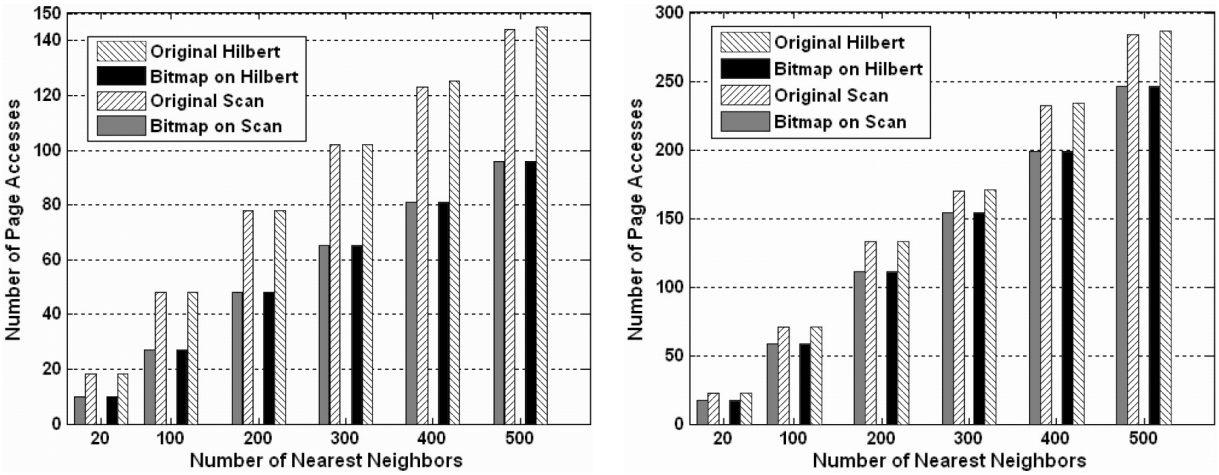
(b) # of Page Accesses on California Data.

Figure 45. Numbers of Page Accesses with Query Composition.

As described in Section 5.3, query composition is an optimization that composites one-dimensional queries with small gaps into a single query to reduce the I/O cost on B+-tree traversals. Experiments were conducted to compare the performance of incremental kNN searches with and without query composition. The results are shown in Figure 45, where the X-axis represents the number of nearest neighbors, and the Y-axis indicates the number of page accesses. The figure revealed that applying the query composition method to both curves dramatically reduced the I/O cost in all queries on both datasets. In both datasets, a kNN search with query composition on Hilbert curves reduced more than 75% of the page accesses when the number of nearest neighbors was greater than 20 compared to the original search on Hilbert curves. The query composition on Scan curves, although not as effective as for Hilbert curves, reduced more than 70%

of the page accesses when the number of nearest neighbors was greater than 20. When the number of nearest neighbors was 20 in the experiments, in most cases the kNN queries could be answered in a couple of iterations, with each querying only a few cells on the B+-tree, so query composition cannot improve performance as much as in larger queries. Based on this set of experiments, it can be concluded that the query composition technique greatly improves the performance of incremental kNN searches, especially for queries with a large number of nearest neighbors, and performs better in Hilbert curves than in Scan curves.

5.4.2.3 Bitmap



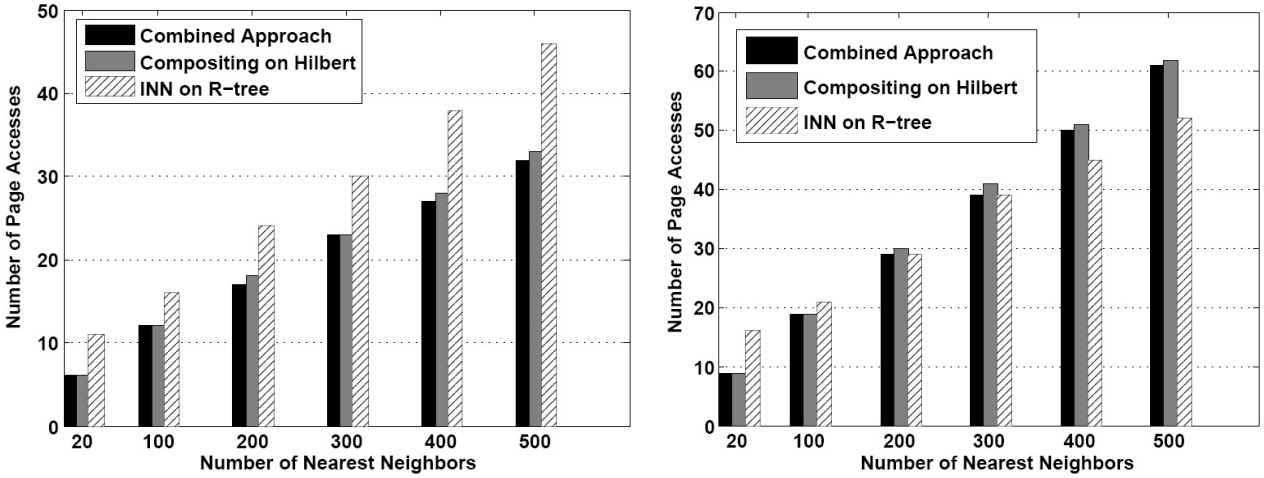
(a) # of Page Accesses on City of Oldenburg Data. (b) # of Page Accesses on California Data.

Figure 46. Numbers of Page Accesses with Bitmap.

The bitmap optimization was applied in this set of experiments in order to reduce the number of cells to be queried on B+-trees. The numbers of page accesses between the original kNN search and that with bitmap optimization are compared in Figure 46, where the X-axis presents the number of nearest neighbors and the Y-axis indicates the number of page accesses. The results showed that the performance of an incremental kNN search with bitmap was consistently better than the original search on both Hilbert and Scan curves. However, unlike the multiple SFCs and query composition optimizations, the bitmap technique reduced the numbers of page accesses differently on the two datasets. Interestingly, it performed better for the City of Oldenburg dataset, which is more uniformly distributed, than for the California dataset. Examining the processing of these kNN queries revealed that because of the skew distribution of the California dataset, some areas were very dense while others were sparse. When the query point was located in a dense area, the bitmap could not filter many empty cells to improve the performance. On the other hand, once the query point was located in a sparse area, the empty cells removed by the bitmap might sometime separate a

cluster of consecutive cells into several pieces, thus increasing the number of tree traversals. The bitmap approach therefore did not reduce page accesses as much in the California dataset as it did in the Oldenburg dataset. An interesting feature of bitmap, as these results showed, is that it improves the query performance on both Hilbert and Scan curves equally, unlike the query composition approach. This is mainly because after removing the empty cells, the queries are more likely to have the same number of clusters under different SFCs for queries on their B+-trees.

5.4.2.4 Optimizations vs. INN on R-tree



(a) # of Page Accesses on City of Oldenburg Data. (b) # of Page Accesses on California Data.

Figure 47. Numbers of Page Accesses of INN on R-tree.

We conducted a set of experiments that combined the above three optimizations to observe the performance gain. The results showed that the I/O performance of the combined approach was only slightly better than the query composition. This indicates that the query composition has mostly optimized the query performance in the proposed method. Comparisons between these optimizations and the Incremental Nearest Neighbor search (INN) on R-tree [7] were made to evaluate the query performance in two different multidimensional access frameworks. In these comparisons, the R-trees were constructed using the same fanout as the corresponding B+-trees. The results of these comparisons are illustrated in Figure 47. On the City of Oldenburg dataset, the INN on R-tree exhibited at least 30% more I/O cost than the NN search with combined optimizations and the query composition on Hilbert curve. The NN search with combined optimizations was only outperformed by the INN on R-tree for the California dataset when the number of nearest neighbors was more than 300. These results suggest that the proposed NN search on SFC and B+-tree benefits more in relatively small kNN queries, because the access method based on B+-tree avoids

overlap among the index nodes, while the R-tree needs multiple traversals for a small search range. Another conclusion, which could be drawn by comparing the results on the two datasets, is that INN on R-tree is suitable for skewed data (the California dataset), because the SFC applied in the experiments evenly divides the space. This limitation can be relieved if hierarchical SFCs are used.

5.5 Conclusion

This chapter describes an efficient approach for incremental kNN searches based on space-filling curves and the B+-tree. This approach uses a priority queue and a simple distance estimation to perform a best-first-search on the minimum set of cells. Three optimization techniques, namely multiple SFCs, query composition, and bitmap, are proposed to improve the spatial query efficiency by reducing the number of page accesses on the B+-tree. Experiments on real datasets showed that both Hilbert and Scan curves have linear scalability for the number of nearest neighbors in the proposed incremental kNN search. The proposed approach exhibited significant I/O improvement against the naïve kNN approach on SFC. The results also demonstrated that the three optimization techniques effectively reduced the I/O cost of kNN searches. Of the optimizations tested, query composition achieved the best performance improvement.

Future research could focus on applying the three optimizations on other spatial operations, such as range queries and spatial joins, to examine their effectiveness. Also, the proposed kNN search algorithm could be extended to handle nearest neighbor searches on moving objects and high dimensional datasets.

Chapter 6. CONCURRENT LOCATION MANAGEMENT ON MOVING OBJECTS

Based on the spatial access framework with B-tree and SFC, a concurrency control protocol for moving object databases, namely Concurrent LocAtion Management (CLAM), is designed by integrating the link-based approach and lock-coupling component. Spatial location updates and range queries are efficiently protected by this protocol.

6.1 Preliminaries

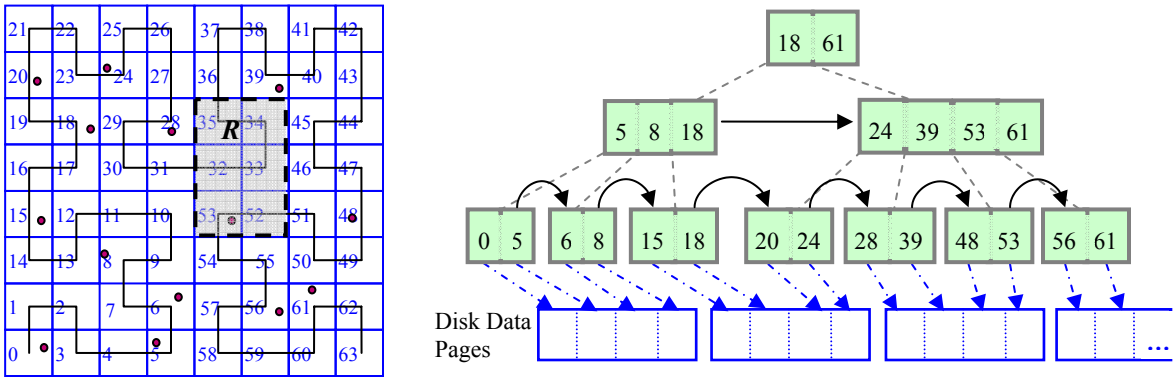
6.1.1 Problem Formulation

In order to propose appropriate solutions for the concurrent location update problem, the application environment has to be described. In this problem, the data access method is based on SFCs and B^{link}-trees (a variant of the B+-tree, as shown in Figure 48) [29]. All the multidimensional data are mapped into equal-sized cells via SFC, associated with their corresponding cell *ID*s. The resolution of grid cells can be determined by the data distribution and disk page size. Then a B^{link}-tree is used to index these SFC values and to maintain the pointers to the data pages. The B^{link}-tree was proposed for link-based concurrency control on B+-trees, and can be easily implemented based on the B+-tree. The location management operations based on this data access method are defined in the following.

The execution of a **range query** returns a set of data objects that overlaps with a given range at the time the query is finished. Formally specified, the input of a range query is a *d*-multidimensional region (query window) *R*, a *d*-dimensional dataset *S*, and the corresponding spatial index *I*. The output of this range query is a set of data points (o_1, o_2, ..., o_n) covered in *R* within *S*. The output data should be valid at the committing time.

A **location update** operation inputs both the old position and new location of a *d*-dimensional data point *o*, as well as a *d*-dimensional dataset *S* and the corresponding spatial index *I*, and outputs the updated *I* and *S*. This operation contains two sub-tasks: delete the old position and insert the new location. An **insert** operation will add a new *d*-dimensional data point. A **delete** operation will remove an existing data point. Both need to update the index if necessary. The execution of a location update should not affect the results

of a query before this update finishes, but has to be reflected in the results of queries after this update commits.



(a) A dataset mapped on Hilbert curve. (b) Corresponding B^{link}-tree.

Figure 48. A Point Dataset with Hilbert Curve Mapping and the Corresponding B^{link}-tree.

Several assumptions are made as follows to give detailed descriptions to this problem.

1. The multidimensional dataset contains only points, which means each data record is treated as a single point in a multidimensional space.
2. The space-filling curves are applied to provide the global cell order, and a standard B^{link}-tree is used as the one-dimensional access method (see Figure 48). Each leaf node of the B^{link}-tree contains the cell *ID*s and the pointers to the data pages that store the corresponding objects.
3. The read/write operation of a node from/to disk is an atomic action. There exists a lock management module to maintain the requested locks and check the operation compatibility.

These assumptions are reasonable in real spatial applications, and they add constraints to the problem formulation.

The goal of the concurrency control protocol is to achieve serializable isolation and data consistency. Serializable isolation means that concurrent operations can achieve the same results as they are sequentially and separately executed. Data consistency refers to the requirement that the query can securely retrieve the data that are consistent with the valid database state.

6.1.2 Observations

The data access model based on SFC and B-tree family leaves spaces for efficiency improvement of concurrent spatial operations. Observations regarding the protocol design and potential performance

improvements are discussed as follows.

Concurrency Control on SFC - In the SFC-based spatial index, the reorganization of B^{link}-trees occurs when inserting a new object into an empty cell, or when one cell becomes empty. Concurrency control is essential for multiple-user environments because updating either index tree nodes or data pages needs exclusive access to keep the data and query results consistent. The link-based concurrency control for B^{link}-trees inserts and deletes entries in a way that the reader can always fetch the valid data by following the left-to-right links to the node with the corresponding key range without placing any locks [27]. This approach is efficient, but cannot be directly applied for spatial queries. Because one such spatial query may require multiple traversals on the B^{link}-tree, a secure protection mechanism has to be devised to keep the searched area unaltered until all the searched objects are returned. For example, the range query *R* in Figure 48(a) covers cells 32, 33, 34, 35, 52, and 53. It requires two B^{link}-tree range searches, cell cluster 32 to 35 and cell cluster 52 to 53. Therefore, not only the indexed cell 53, but also all the other five empty cells (32, 33, 34, 35, and 52) must be protected from other update operations. Apparently, the link-based approach is not adequate to assure this kind of protection, as it can only protect the indexed cells among updates. Therefore, lock-coupling techniques have to be incorporated. Once the update operation needs to insert/delete a cell or modify a data object, it has to assure that the cell/object is not located within the ongoing search region. Note that not only the non-empty cells, but also the empty cells in the search area have to be locked, because in case a new object is inserted into an empty cell within the search range, it will invalidate the final search results.

Lock Efficiency - In the spatial access method based on B^{link}-trees and SFCs, location update operations can be handled in different manners based on the SFC cells that contain the old location or new location. The index tree modification only occurs when inserting a new object into an empty cell, or when a cell becomes empty. Otherwise, only the corresponding data pages need to be protected and modified. Obviously, if these distinct scenarios are not handled separately, the concurrency control protocol will need to lock all the B^{link}-tree leaf nodes and data cells that will be accessed during the update, and release them in the very end, which could significantly degrade the system throughput. Therefore, for performance consideration, these scenarios need to be respectively treated, so that unnecessary locks can be quickly released to increase the concurrency level.

6.2 Concurrent Spatial Operations

To explain the algorithms for concurrent operations in CLAM, the fundamental locking mechanism will be

illustrated in the following subsection, followed by the detailed location update operation and range query algorithms.

6.2.1 Lock Map

Table 8. Types of Locks and Their Compatibility in CLAM.

		Cell-level		**Node-level**
		Write-lock	**Read-lock**	**Write-lock**
Cell-level	**Write-lock**	***Exclusive***	***Exclusive***	***N/A***
	Read-lock	***Exclusive***	***Compatible***	***N/A***
Node-level	**Write-lock**	***N/A***	***N/A***	***Exclusive***

In the proposed concurrent spatial operations in CLAM, two levels of locks are used (shown in Table 8). One is **node-level lock**, which is placed on index tree nodes. The node-level locks, requested only by update operations, are all write-locks. The other is **cell-level lock**, which includes both read-lock and write-lock and will be requested on SFC cells by all the spatial operations. As discussed in Section 6.1, for spatial queries, not only non-empty cells, but also empty cells will need to be read-locked, since they are not allowed to be modified during the query process. Therefore, an auxiliary **lock map** structure, in addition to the index tree, is applied in the proposed framework to dynamically maintain cell-level locks. The concurrent spatial operations have to check the corresponding records in the lock map before placing cell-level locks. Each cell maintains a counter for its current read-locks, and uses -1 to indicate the write-lock. A queue also is used by each cell to store the pending cell-level locks, so that these waiting processes can be notified and activated once the cell is available. After an operation checks the compatibility of the cell-level locks, if the cell is currently unavailable (i.e., has incompatible locks), this operation will be recorded in the pending queue. A lock map example is illustrated in Figure 49, where the pending queues of two cells are shown. As the lock map will be frequently accessed in this concurrency control framework, it can be implemented using a hash table that is sliced based on the spatial distribution to avoid causing a performance bottleneck. In addition, sophisticated compression techniques can be applied to reduce the space requirement. Note that only cell-level locks need to access the lock map, because a cell-level lock will not conflict with a node-level lock in CLAM.

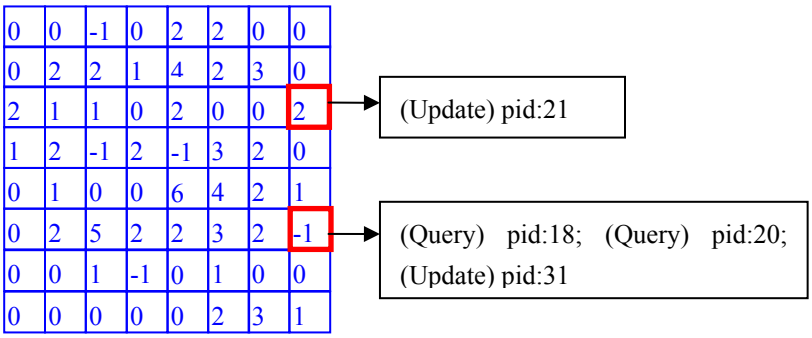


Figure 49. A Lock Map Example.

6.2.2 Location Update

In order to protect the search operations from the interference of update operations, the location update operations need to check both the node-level locks on the B^{link}-tree leaf nodes, and the cell-level locks on the data cells. On the other hand, the query operations have to check the write-locks on the cells. For cell-level locks, the write-lock from update operations and the read-lock from read operations are exclusive to each other. In case the cells for updating have been locked by another operation, the update operations on these cells have to wait until they can successfully write-lock them. Specifically, the concurrent location update is performed as follows.

A location update operation first deletes an existing object, and then inserts a new object with the same object identifier (*ID*) into the data page. In this operation, there will be two exclusive scenarios:

1. The old location is not the last item in its cell, and the new location is located in a cell corresponding to an entry in the B^{link}-tree, thus the index tree will not need to be modified.
2. The new location is located in a cell that is not indexed in the B^{link}-tree (i.e., an empty cell), or, the old location is the last item in its cell, thus the corresponding nodes in the B^{link}-tree have to be locked and modified.

To perform an update operation, the **first phase, identification,** is to locate the corresponding leaf node that contains or will contain the cell of the new location, as well as the leaf node that contains the cell of the old location. To pinpoint the leaf nodes, the SFC values of the locations are calculated based on the specific curve. Then the B^{link}-tree is traversed to find the entry corresponding to the cell, in a way similar to the fundamental read operation on B^{link}-trees, except that this process needs to cache the traversed path. The cached path can help locate the parent nodes in case of node split or merge. By the end of this identification phase, node-level locks are requested on the leaf nodes that have been located.

Algorithm Location Update (*old_loc, new_loc*, *T*, *LM*)

Input: *old_loc:* location to be removed, *new_loc*: location to be inserted, *T:* B^{link}-tree, *LM:* Lock map,
Output: *T:* Updated B^{link}-tree.

//Identification

1. *c_old = SFC_map(old_loc)*; //determine the cell contains *old_loc*
2. *c_new = SFC_map(new_loc)*; //determine the cell contains *new_loc*
3. *n_old = T.traverse(c_old)*; // locate the leaf which contains *c_old*
4. *n_new = T.traverse(c_new)*; // locate the leaf which contains *c_new*
5. *T.writeLock(n_new & n_old);* // request node-level locks at one time

//Modification

6. *LM.writeLock(c_old* and *c_new)*; // request write-lock on cell *c_old* and *c_new*
7. If (*c_new.size > 0*) // *c_new* is not empty
8. *T.unWriteLock(n_new)*; // release lock on node *n_new*
9. If (*c_old.size > 1* or *c_old == c_new*) // no need to delete *c_old*
10. *T.unWriteLock(n_old)*;// release lock on node *n_old*
11. If (*c_old.size == 1* and *c_old != c_new*) // need to delete *c_old*
12. *n_old.removeEntry(c_old); //*delete entry for *c_old* from *n_old*
13. If (*n_old.underflow == true*)
14. *n_old.merge()*;
15. *T.unWriteLock(n_old)*; // release node-level write-lock on *n_old*
16. If (*c_new.size = 0*) // *c_new* is empty
17. *n_new.addEntry(c_new)*; //add entry for *c_new* into node *n_new*
18. If (*n_new.overflow == true*)
19. *n_new.split()*;
20. *T.unWriteLock(n_new)*;// release node-level write-lock on *n_new*
21. *PageDeletet(n_old.entry(c_old), old_loc)*; //remove *old_loc* from the data page that contains cell *c_old*
22. *PageInsert(n_new.entry(c_new), new_loc)*; //insert *new_loc* to the data page that contains cell *c_new*

//Commitment

23. *LM.unWriteLock(c_old* and *c_new)*; // remove cell-level write-locks
24. Return *T*;

Algorithm 13. Concurrent Location Update.

The **second phase, modification,** is to access the actual data page and update its content. This phase contains two conditional branches, corresponding to the two scenarios determined by the number of objects in the data cells. The **first** branch will request write-locks at one time on the cells to be modified. The request of the write-locks needs to access the lock map to determine whether this process should continue or be suspended. If the cell in the lock map has the value 0 (i.e., not locked), this operation will mark it as -1 (write-locked) and continue; otherwise, it will enqueue its process *ID* and wait. After locking the corresponding SFC cells, the node-level locks requested in the identification phase will be released, since

no nodes will be modified. The algorithm will then update the data pages by inserting the new location and deleting the old position. The **second** branch, which occurs when the object moves from a single-item cell or relocates to an empty cell, requires keeping node-level locks on the B^{link}-tree. In this branch, the cells that contain the new or old location will be write-locked, and then the leaf node that does not need to be changed, if any, will be unlocked. In this way, only the items that need to be updated will be securely locked. After releasing the unnecessary node-level locks on the nodes that do not need to be changed, if the cell that contains the old location will not have any data object after the update, the corresponding entry in the leaf node will be deleted. If this leaf node has capacity more than or equal to the minimum node capacity, the delete process is accomplished. Otherwise, a node merge operation similar to the concurrent merge in B^{link}-trees has to be performed to restructure the tree. In this way, the delete operation assures that other concurrent operations can obtain valid results. On the other hand, if the new location is in an empty cell, a new entry will be added to the corresponding leaf node. If this leaf node for insertion has sufficient space for a new entry, the cell will be added to this node directly, and then the write-lock will be released. Otherwise, the leaf node will be split into two nodes by adding a new leaf node as the right neighbor of the original node, following the concurrent split operation in B^{link}-trees. A propagation split will be performed if necessary, following the cached path. After modifying the B^{link}-tree and releasing all the node-level locks, the operation then updates the actual data pages.

The **final phase, commitment,** releases the write-locks on the cells and returns the updated B^{link}-tree. The detailed concurrent location update algorithm is described in Algorithm 13.

6.2.3 Range Query

Given a search range *R*, a range query returns all the objects that are covered by *R*. This operation requires only read-locks. To execute a range query, the spatial query range *R* will be mapped to a set of one-dimensional ranges using an SFC. After that, these one-dimensional ranges will be queried on the B^{link}-tree. Each one-dimensional query is executed as the fundamental concurrent search operation on the B^{link}-tree, whereas the major difference is that all the cells that overlap with *R* will be read-locked before being scanned, regardless of whether they are empty or not. For example, in the range query *R* shown in Figure 48, the read-locks will be placed on cells 32, 33, 34, 35, 52, and 53, before retrieving any of them. Therefore, the corresponding records in the lock map will be checked, and if the cell is available, its read-lock counter will be incremented; otherwise, the corresponding process *ID* will be inserted into the pending queue. All these read-locks will be released only when the entire range query is complete. This locking strategy assures that these cells will not be altered during the entire process of the spatial range

query. The detailed concurrent range query algorithm is presented in Algorithm 14. In the **initiation** step (line 1-3), the cells overlapped with the query range will be identified. In the **tree traversal** stage (line 4-10), for each consecutive cell cluster (e.g., cells 32-35 in query *R*), the tree will be traversed from the root to leaf, and the corresponding data cells will be read-locked before retrieving the data pages. Once all the indexed cells that overlap with the query range have been accessed, in the **commitment** step, exact results are returned and all the requested cell locks are released.

Algorithm RangeQuery (*R*, *T*, *LM*)
Input: *R:* Query range, *T:* B^{link}-tree, *LM:* Lock map,
Output: *S*: Set of objects covered by *R*.

//Initiation
1. *S* = {}; // initiate the result set
2. *L*={}; // initiate locked set
3. *SC* = *SFC_map(R)*; // determine the cells that overlap with *R* using SFC

//B^{link}-tree traversal
4. For each cell cluster *C* in *SC*
5. *n* = *T.traverse(C)*; // locate the left-most leaf node which overlaps with cell cluster *C*
6. While (*n.minKey* <= *C.maxKey*)
7. *P* = *n.entries* ∩ *C*;
8. *LM.readLock(P)*; // request read-lock on cell set *P*
9. *L* = *L* + *P*; // record the locks
10. *S* = *S* + *PageRetrieve(n.entry(P))*; // retrieve objects inside *P*

//Commitment
11. *S* = *S* ∩ *R*; // filter the objects outside of *R*
12. *LM.unReadLock(L)*; // release the cell-level locks
13. Return *S*;

Algorithm 14. Concurrent Range Query.

6.3 Correctness of Concurrency Control

In the proposed concurrent spatial operations, the three requirements of concurrency control protocols can be achieved, namely, serializable isolation, data consistency, and deadlock-freedom.

Serializable Isolation - The node-level locks isolate the concurrent location update operations, because these operations place node-level write-locks in a bottom-up manner on the B^{link}-tree when reconstruction is required. On the other hand, the cell-level locks serialize the concurrent update and search operations on data pages. Even though the isolated order of operations may not be exactly in the same sequence as they

started, it is regarded as valid in concurrency control protocols because it is inevitable to suspend or restart some operations.

Data Consistency - The spatial concurrent operations can be assured to securely retrieve valid results consistent with the current status. For instance, suppose a range query *R* and an update operation *U* occur simultaneously with cell *C* as the common resource, the results of *R* will reflect *U* only when the read-lock is placed after the write-lock on *C*, which means the new data in *C* is inserted into the dataset before *R* is accomplished. Furthermore, in case the reconstruction caused by *U* is performed while the traversal of *R* is in process, as described in the B^{link}-tree search algorithm, *R* can always follow the down-and-right links to reach the leaf node that currently contains *C*. The only situation where *U* will not impact the results of *R* is that the write-lock on *C* is successfully placed after all the read-locks from *R* are released, which means the data in cells *C* is inserted after the commitment of *R*. Two concurrent update operations can guarantee the final results are consistent with the current dataset, because they only can be processed with the cell-level write-locks and bottom-up node-level locks successfully granted, which prevents any conflict.

Deadlock Free - The proposed operations will not cause additional deadlocks, because range queries need to access multiple cells, and they only read-lock these cells. Meanwhile, each location update operations place cell-level write-locks at one time, which will not cause deadlocks with search operations. In an update operation, all node-level locks will be placed at one time, and these locks will either release, or expand upward or rightward during this process. Furthermore, each update operation will request cell-level locks after write-locking the corresponding leaf nodes. Therefore, there will not be any two update operations that hold the resources required by each other and wait for each other indefinitely. A detailed proof by examining all possible combinations of these concurrent operations is given as follows.

Proof:

We only need to prove that a location update will not interfere with any concurrent operations, because a range query can never affect any other range queries. There are two conditional branches in the proposed location update operation. Following each branch, a location update may occur simultaneously with a range query and two types of another location update on common cells.

Branch 1: Without index modification

In this branch, the cell that contains the old location will not be empty after the update, and the cell that will encompass the new location exists before the update. At first, node-level locks will be placed together at

one time on leaf nodes *n_new* and *n_old*. Cell-level write-locks then will be requested together at one time on cell *c_old* and *c_new* using a lock map before unlocking the leaf nodes and before updating the data pages. If a **range query** starts during this update and covers cell *c_old* or *c_new*, it needs to obtain a read-lock on cell *c_old* or *c_new* before actually reading the data pages. Since this read-lock is exclusive to the write-locks requested by the location update, the range query will have to wait if the write-locks have been placed, or will keep the location update waiting if the write-locks have not been placed. Therefore, the intermediate status of the location update will not be retrieved. Furthermore, because the write-locks on cell *c_old* and *c_new* are requested together as an atomic action, the location update will either wait without holding any locks or proceed after obtaining all the requested locks. There will be no deadlock during this process.

If another **location update** (say, *U**) in **branch 1** or **branch 2** starts during this update and affects cell *c_old* or *c_new*, it needs to request cell-level write-locks on cell *c_old* or *c_new* before actually modifying the data pages. Since these write-locks are exclusive to the write-locks requested by the original location update, *U** will have to wait if the write-locks have been placed by the original update, or will keep the original update waiting if the write-locks have not been placed. Therefore, inconsistent status of the index and data will not be retrieved. In this branch, all the cell-level locks/node-level locks in each operation are requested as an atomic action, so there will be no deadlock. No other operations are related to this location update from the aspect of concurrency control.

Branch 2: With index modification

The cell that contains the old location will not be empty after the update, and the cell that contains the new location exists before the update. In this branch, the cell-level write-locks will be requested on cell *c_old* and *c_new*. The node-level locks will be placed on the B^{link}-tree leaf nodes *n_old* and *n_new* at the beginning, and be kept on the nodes needing to be modified till the end of the operation. In case a **range query** starts during this update process and covers cell *c_old* or *c_new*, it needs to obtain a read-lock on cell *c_old* or *c_new* before actually reading the data pages. Since this read-lock is exclusive to the write-locks requested by the location update, the range query will have to wait if the write-locks have been placed, or will keep the location update pending if the write-locks have not been placed. Therefore, the intermediate status of the location update will not be accessed. Furthermore, because the write-locks on cell *c_old* and *c_new* are requested together as an atomic action, the location update will either wait without holding any locks or proceed after obtaining all the requested locks. There will be no deadlock during this process.

If another **location update** *U** in **branch 1** starts during this update and affects cell *c_old* or *c_new*, it needs to request cell-level write-locks on cell *c_old* or *c_new* before actually modifying the data pages. Since these write-locks are exclusive to the write-locks requested by the original location update, *U** will have to wait if the write-locks have been placed by the original update, or will keep the original update waiting if the write-locks have not been placed. Therefore, inconsistent status of the index will not be accessed. Similarly, since all the write-locks in each operation are requested as an atomic action, no deadlock will occur in this case. If *U** is a **location update** in **branch 2** that starts during this update and affects tree node *n_old* or *n_new*, it needs to request node-level locks on *n_old* or *n_new* before requesting cell-level locks and before actually modifying the tree nodes and data pages. Since these locks are exclusive to the node-level locks requested by the original location update, *U** will have to wait if the node-level locks have been placed by the original update, or will keep the original update waiting if these locks have not been placed. Therefore, inconsistent status of the index and data will not be retrieved. Similarly, since all the node-level locks or cell-level locks in each operation are requested together as an atomic action, no deadlock will occur in this branch. Furthermore, as the cell-level locks are placed after the node-level locks have been obtained, if any location update needs to modify the corresponding leaf node, it can always retrieve the valid nodes. All other operations are not related to this location update from the aspect of concurrency control.

To summarize the proof, all the possible combinations of concurrent spatial operations and their conditional braches have been thoroughly examined. All of them are shown to meet the requirements of serializable isolation, data consistency, and deadlock-freedom. This proof is complete.

Q.E.D.

6.4 Experiments

To evaluate the performance of the proposed concurrent spatial operations, sets of experiments on real datasets have been conducted by comparing the throughput (number of operations processed in a time unit) among different concurrency control protocols, as shown in Figure 50. The two real datasets used in the experiments are 6,000 road network nodes in the City of Oldenburg and 62,000 points of interest in California, both from [22], as shown in Figure 51. The road nodes for the road network in the City of Oldenburg, mapped using a Hilbert curve with order 5, are relatively uniformly distributed with about 40% empty cells. On the other hand, the dataset for points of interest in California, mapped using a Hilbert curve with order 8, has a rather skewed distribution with about 80% empty cells. Based on the SFCs, a B^{link}-tree

with height of 3 was built on the City of Oldenburg dataset, and a tree with height of 4 on the California dataset, respectively. The fanout of the B^{link}-trees built in experiments is 32.

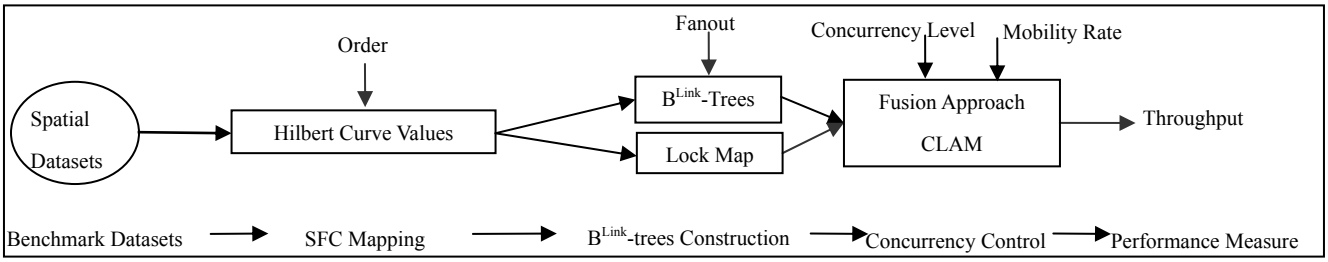
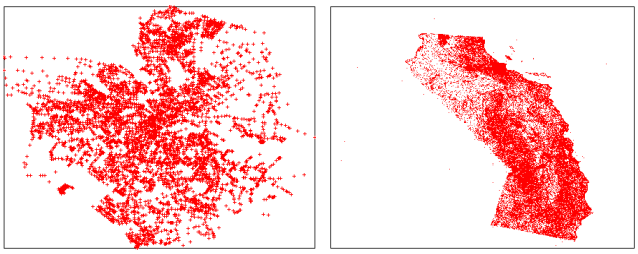


Figure 50. Experiment Flow.

In the experiments, the range queries were created by randomly selecting the center points and setting the window size as 5% of the whole data spaces, and each location update was generated by randomly assigning an existing object as a start point and half length of the cell width as the moving pace. The **concurrency level** (the number of operations simultaneously processed in one batch) of the operations, and the **mobility rate** (the percentage of location updates in the entire operation set) of the moving objects varied in the experiments as the parameters to simulate different application environments. The processing time of operations was used to evaluate the performance of the proposed concurrent operation framework. A **fusion** concurrency control approach, which applies the link-based locking on the B^{link}-tree and the lock-coupling locking on the lock map, was designed and implemented in the experiments for comparing with the proposed concurrent protocol. Different from CLAM, this fusion protocol requests the locks at the beginning of a location update and releases them at the end. This fusion approach applies the link-based locking on the index tree, which has been shown to have fewer read/write conflicts with less maintenance overhead than lock-coupling protocols; therefore it can achieve higher throughput than the pure lock-coupling approach. Note that the proposed approach was compared to a sophisticated method, fusion, to demonstrate the advantage against an advanced method.



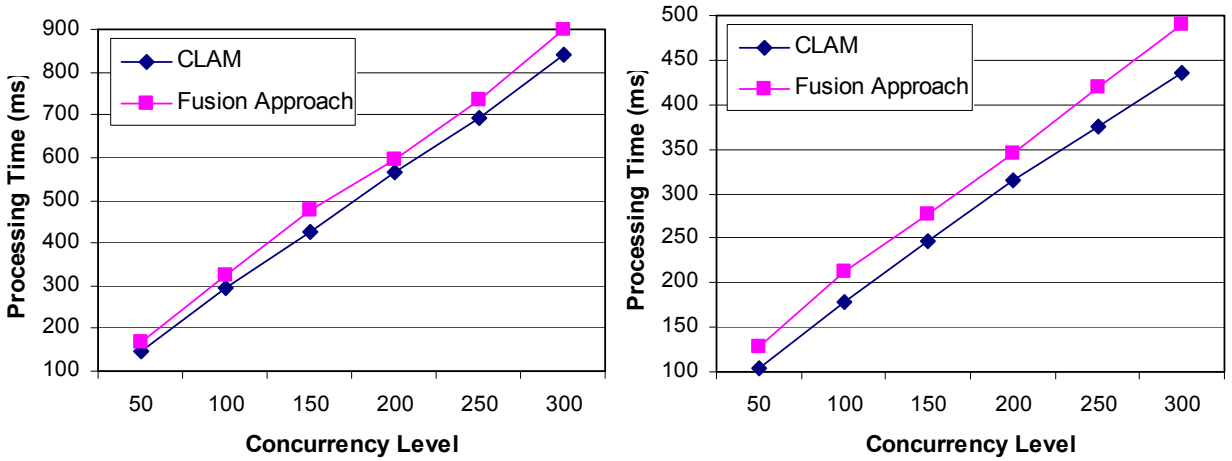
(a) City of Oldenburg. (b) California places.

Figure 51. Experiment Datasets.

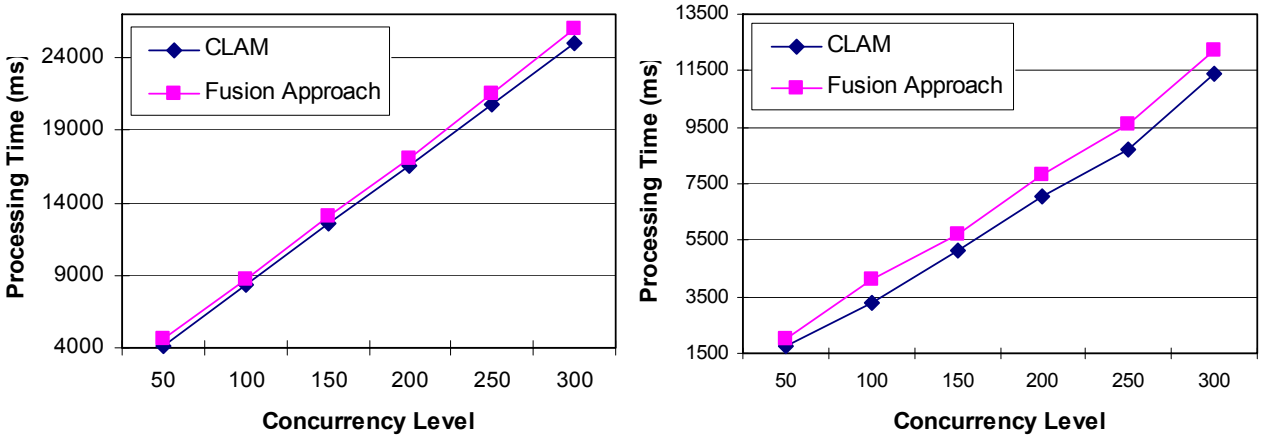
Two sets of experiments are described in the following subsections. The first set of experiments shows the efficiency and scalability of the proposed location management on Hilbert curves by examining the

throughput under different concurrency levels. The second set of experiments demonstrates the impact of mobility rate on throughput. The comparisons between the fusion concurrency control approach and the proposed CLAM framework were made in both sets of experiments.

6.4.1 Throughput vs. Concurrency



a) Oldenburg data (with mobility rate = 30 (left) and 70 (right)).



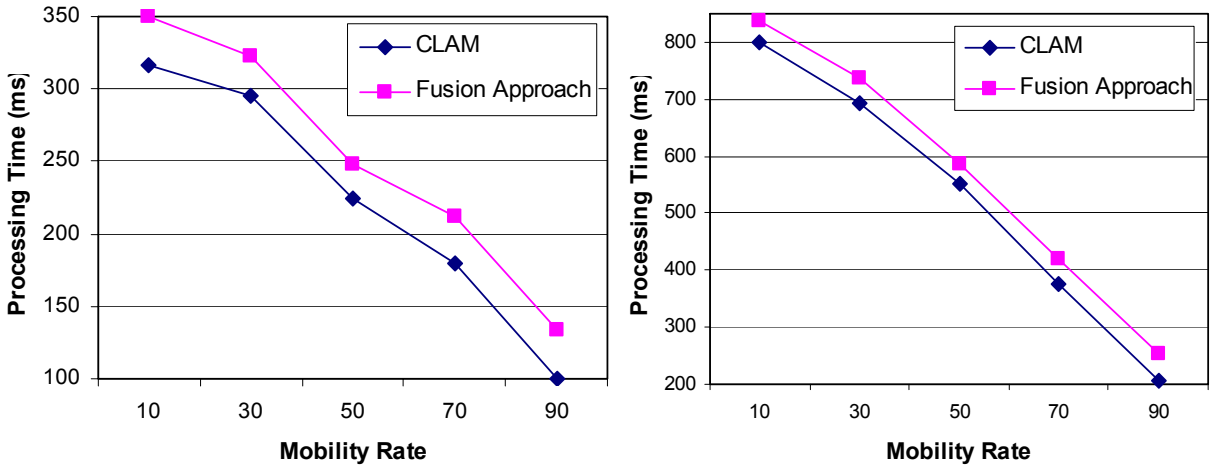
b) California data (with mobility rate = 30 (left) and 70 (right)).

Figure 52. Processing Time under Different Concurrency Levels.

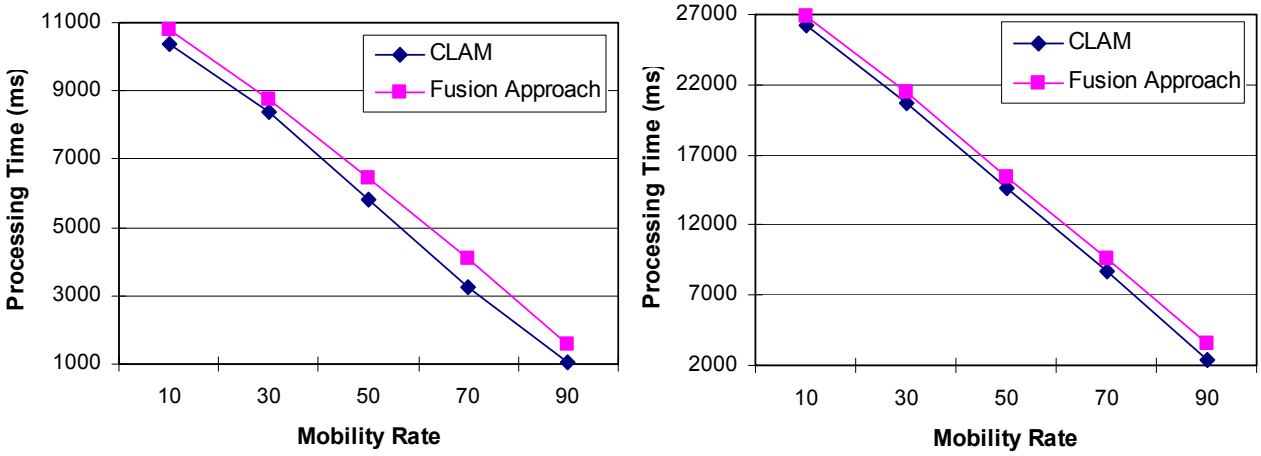
The first set of experiments compares the processing time between concurrent location management of CLAM and the fusion approach under different concurrency levels, in order to determine how the concurrency workload affects the system throughput. The processing time of the concurrent operations was collected with the mobility rate sets as 30 percent and 70 percent of the whole operation set. Similar trends of the processing time also have been observed under different mobility rates.

Figure 52 shows the processing time of the concurrent operations with various concurrency levels, in which the X-axis represents the concurrency levels and the Y-axis indicates the processing time in milliseconds. As illustrated in Figure 52, in both datasets, the processing time of both concurrency control approaches generally increases linearly with regard to the concurrency level. In this set of experiments, CLAM performs 10-20% better than the fusion approach. Furthermore, the higher the concurrency level, the larger the gap between these two approaches. This is reasonable because more concurrent operations in a batch could cause more read-write conflicts. Consequently, the processing time saved by efficiently releasing unnecessary locks, as employed in CLAM, will become more significant.

6.4.2 Throughput vs. Mobility



a) Oldenburg Data (with concurrency = 100 (left) and 250 (right)).



b) California data (with concurrency = 100 (left) and 250 (right)).

Figure 53. Processing Time under Different Mobility Rates.

This set of experiments compares the processing time between CLAM and the fusion concurrency control

approach with different mobility rates. The processing time of the concurrent operations was collected with the concurrency level set at 100 and 250 correspondingly. Similar trends of the processing time have been observed under different concurrency levels.

Figure 53 shows the processing time of the concurrent operations as the mobility rate increases, where the X-axis indicates the mobility rates and the Y-axis represents the processing time in milliseconds. As observed from Figure 53, in both datasets, the processing time of both approaches linearly decreases when the mobility rate increases. This is because a range query usually needs to access as many data pages as the number of SFC cells it covers, while a location update only needs to access at most two data pages. In this set of experiments, CLAM performs significantly better than the fusion approach. Furthermore, the advantage of CLAM against the fusion approach becomes more significant when the mobility rate increases. For example, in the Oldenburg dataset with the concurrency level of 100 (left figure in Figure 53(a)), the fusion approach takes 10% longer than CLAM to process the operations when the mobility rate is 10, while it takes 30% longer time than CLAM when the mobility rate is 90. This is because CLAM optimizes the locking strategy in the location update operation (Algorithm 13). When there are more location update operations, CLAM is expected to benefit more from its optimizations.

These experimental results show that the proposed concurrent location management approach exhibits scalable performance when the concurrency level increases or when the mobility rate decreases. It outperforms the fusion concurrency control approach with different parameter settings, especially in the high mobility situation. This indicates that the proposed concurrency framework can achieve optimal performance in general, and is suitable to manage frequent concurrent location updates and queries.

6.5 Conclusion

This work proposes a concurrent location management framework, CLAM, for efficiently handling of moving objects. CLAM provides adequate protection for concurrent location update and search operations to achieve serializability, consistency, and deadlock-freedom. The correctness of CLAM is formally proved by completely examining all the possible scenarios during the concurrent operation processing. Experimental results on real datasets have demonstrated that the performance optimizations in CLAM are effective. Further efforts could focus on extending CLAM to support other spatial operations, such as range aggregation and nearest neighbor search. Meanwhile, adopting the design of CLAM to other moving object access methods based on B+-trees offers attractive possibilities for concurrent moving object management.

Chapter 7. CONCURRENT SPATIAL CONTINUOUS QUERIES ON B-TREES

This chapter proposes a concurrent continuous monitoring approach based on the multidimensional indexing structure with general SFCs and the B^{link}-tree. This approach ensures serializable isolation, consistency, and deadlock-freedom on scalable continuous query processing and location update by extending CLAM to multiple indices.

7.1 Preliminaries

Before presenting the concurrent continuous query processing in detail, we introduce the overall design of this proposed framework. In this framework, serializable isolation is provided 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 commission time. The access framework designed for scalable continuous query processing captures the current locations of the objects and queries. In addition, the proposed concurrency control protocol can be conveniently integrated with other motion-based indexing structures, such as the B^x-tree and BB^x-tree, to concurrently access past and future status.

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 and described 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 design of indexing structure and its concurrency control protocol is presented in the rest of this section.

7.1.1 Index Framework

To process continuous queries with efficiency and scalability, an indexing structure with one B^{link}-tree and two hash tables is applied to index the current locations of both objects and queries. In this framework, SFC divides the space into non-overlapped cells, and maps each object into a particular cell and each query into a set of corresponding cells. Thus the spatial locations of objects/queries can be represented by one-dimensional cell *IDs* in a way that their spatial localities are well preserved. The cell *IDs* of the moving objects then can 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 the 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-tree, is that B-trees facilitate window search while increasing maintenance cost, whereas hash tables provide efficient random access. Hash tables generally require less I/O to access a random entry than B-trees. Furthermore, hash tables are easy to construct and maintain, because they do not require splitting or merging, which are normal operations on B-tree nodes. The performance of hash tables heavily depends on hashing functions. As long as the hashing function distributes the entries evenly into the buckets, its performance is optimized. In the proposed query processing algorithms, there is no window search on the query index involved. In this case, hash tables become a better solution, assuming an applicable hashing function is adopted. The benefit of applying hash tables for moving queries has been validated in the experiments. 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 the hash index, and each entry stores a list of objects covered by a particular query.

A snapshot example of moving objects and continuous queries is illustrated in Figure 54, where a 2D Hilbert curve with order of 3 is applied. There are six queries indicated by letters, and fourteen objects denoted by their cell *IDs* in the example. Figure 55 demonstrates a B^{link}-tree constructed based on the objects in Figure 54. 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 in consecutive pages if there is overflow. Different from standard B-trees, in B^{link}-trees, the tree nodes on the same level are linked from left to right. The

Q-table for the query index corresponding to Figure 54 is shown in Figure 56. 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 57 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 status of a database. The objects in the example are denoted in the form of *O_cellID*.

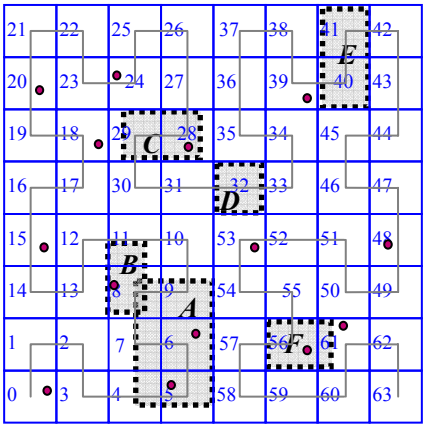


Figure 54. Example of Objects and Queries.

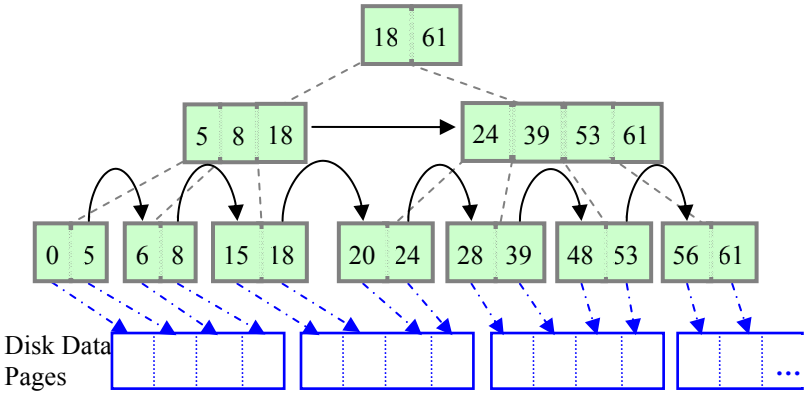


Figure 55. A B^{link}-tree Index for Objects in Figure 54.

In this index 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 cell 53 is moving to cell 32 and covered by 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 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 the 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 the new entries 38 and 39 with *E* inside. The *R-table* is refreshed by inserting *O_39* into entry *E*. A query report simply visits the entry in the *R-table* and outputs the query results.

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 56. Q-table for Queries in Figure 54.

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

Figure 57. R-table for Objects and Queries in Figure 54.

7.1.2 Concurrency Control Protocol

Continuous query processing requires an appropriate concurrency control protocol to ensure the correct results while objects and queries are moving. Taking the scenario in Figure 2 as an example, inconsistent results are caused by incorrect processing sequences. Suppose each movement can be divided into three 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 that contains … $\rightarrow A.D \rightarrow Q.R \rightarrow A.I \rightarrow qR \rightarrow A.R \rightarrow$... will output *null* as the results of *Q* at time t_2, because *A* disappears in the database when *Q* updates its results, and no other update occurs before the query report for t_2. Another inconsistent result set *{B}* of *Q* at t_2 will be returned if the processing sequence contains … $\rightarrow B.R \rightarrow Q.D \rightarrow qR \rightarrow Q.I \rightarrow$…. In this case, 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* has any chance to re-evaluate its results. If a processing sequence contains …$B.R \rightarrow Q.I \rightarrow A.R \rightarrow qR \rightarrow Q.R$…, both *A* and *B* will be output as the results of *Q* at time t_2. 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; 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 the updates and searches can be isolated from each other. The proposed concurrency control protocol provides the following protections.

- Intra-consistency: Consistency on index tree structure;
- Inter-consistency: Consistency among *Q-table*, *R-table* and the B^{link}-tree;
- Deadlock-freedom: No deadlock occurred by accessing multiple indices.

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

To globally manage the locks in all the operations, a lock manager is proposed based on the B^{link}-tree, SFC, and query list. As listed in Table 9, 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 of each other. The reason for differentiating these two sets of lock granules is to allow more concurrent accesses for better throughput. The lock manager maintains all the read-locks and write-locks on cells for objects and queries, in order to prevent phantom access. The locks on the cells for objects are maintained by the Object Lock Map (OLM). On the other hand, 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 58. In each lock map, each cell is associated with an integer to indicate the number of current read-locks granted for the cell. If a cell has been write-locked, -1 will be assigned to that cell in the lock map. In addition, a queue also is used by each cell to store pending cell locks, so that these processes can be notified once the cell is available.

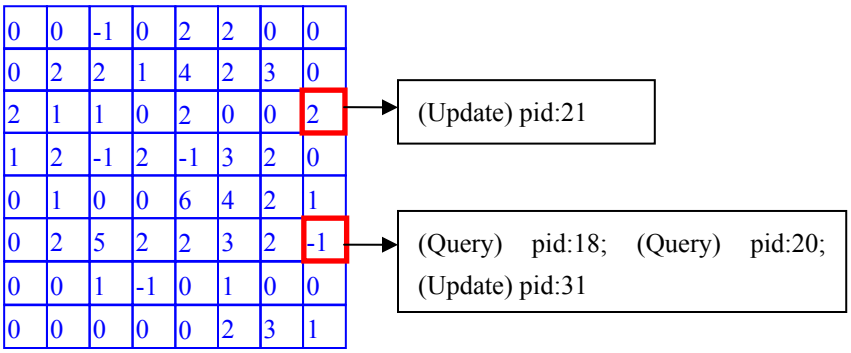


Figure 58. An Example of Lock Map.

In all these locks, read-locks are compatible with each other, while write-locks are exclusive to either read-locks or other write-locks. The tree nodes are protected by the link-based locking strategy, whereas the queries and cells are secured by the lock-coupling strategy. The collaboration among different locks assures the effectiveness of the proposed concurrency control protocol. The spatial operations for continuous monitoring on moving objects integrated with these locks are introduced in detail in the next section.

Table 9. Types of Locks Maintained.

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

7.2 Concurrent Continuous Query

The proposed concurrent access framework based on the B^{link}-tree and SFC supports object movement, query movement, and query report to accomplish the scalable continuous query for moving objects. These operations are designed to be processed concurrently without interfering with each other.

7.2.1 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 the 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: object location update, query search, and result refresh. The details of the algorithm are shown in Algorithm 15.

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 15. Object Movement.

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 known, 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 the 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 the different situations, 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 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 out from, and is added into the entries for the queries that the object is moving to. After all the necessary updates are made 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 the other operations.

Taking the example of object movement in Section 7.1.1, assume the object in cell 53 is moving to cell 32, covered by query *D*. This algorithm first locates the leaf nodes that contain cell 53 or will contain cell 32, and requests write-locks on these nodes. The OLM then requests write-locks on cell 53 and cell 32. The entry for cell 53 is deleted from the leaf node, and a new entry for cell 32 is inserted, before the locks on these leaf nodes are released. In Phase 2, the QLM requests read-locks on cell 53 and cell 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 entry *D* in the *R-table*, and the query lock on *D* and the OLM locks on cell 53 and cell 32 are released.

7.2.2 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 remain 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 16 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 the concurrent object movements retrieve the 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 shared. 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.

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 7.1.1, in which the query *E* is moving towards the 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 cell 39 is retrieved via the B^{link}-tree. The QLM then requests write-locks on cells 38, 39, 40, and 41. Entries for cells 40 and 41 in the *Q-table* remove query *E*. Meanwhile entries for cells 38 and 39 add *E*. By the end of Phase 2, a query write-lock is placed on *E*, and the QLM locks are released. In Phase 3, the object in 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.

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 16. Query Movement.

7.2.3 Query Report

A query report takes a query *ID* and the *R-table* as input parameters, and returns the objects that are covered currently by this given 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 17. 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. By the time of commitment, 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 54 as an example, if a 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 cell 56, is then retrieved. Finally the lock on *F* is released before the commitment.

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 17. Query Report.

7.3 Correctness

The proposed concurrent continuous query processing guarantees serializable isolation, consistency, and deadlock-freedom based on the well-designed concurrency control 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 of the database; **deadlock-freedom** means any combination of the concurrent operations do 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 59 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 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 the horizontal projections of the bars indicate the intersections of the duration of the corresponding locks. The label for each bar shows the particular sub-structure that is locked. Bars with label *BL* are the locks on the B^{link}-tree nodes, while *OL* and *QL* are the locks on OLM cells 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-freedom.

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 commitment. 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 commitment, holds locks on the query window/query results (*QL* and *RL* in Query Movement) before updating the query, and releases these locks until the commitment. 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 concurrent operations. From the above analysis, all these operations lock the target items before accessing them and till the end of processing. This locking strategy guarantees that no conflicting access on common resources could occur among concurrent operations. Therefore, the proposed operations are serializably isolated.

Consistency: The design of the B^{link}-tree, including the write-locks on tree nodes (*BL* in Object Movement) has been proved [56] to assure the consistency of 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 that the intermediate status during the process will not be accessed by other operations. Because the query report locks the query results (*RL* in Query Report) from its initiation to its termination, and the movement operations also lock the query results till their commitments, 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 regarding the current database.

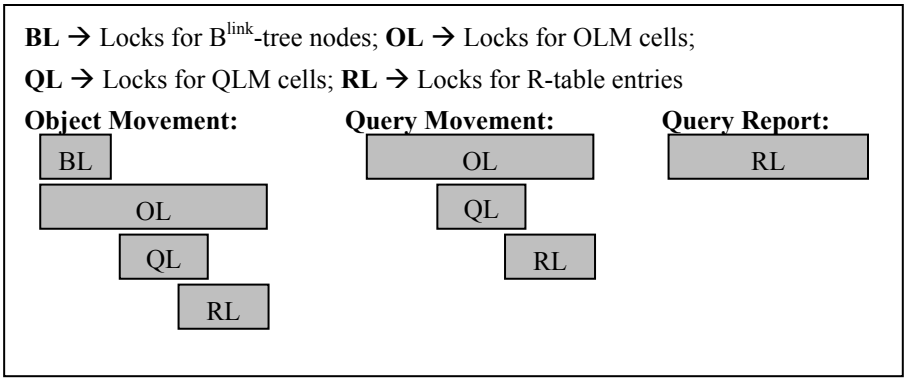


Figure 59. Lock Durations for Concurrent Operations.

Deadlock-freedom: Deadlock-freedom is assured as long as the common sources are not accessed in the opposite order. The B^{link}-tree is deadlock-free internally, as long as the tree node locks (*BL* in Object Movement) are requested according to a global order. The locks in OLM, QLM, or *R-table* (*OL*, *QL* or *RL*

in Figure 59) 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.

The correctness of the proposed concurrency control protocol is clearly shown from the above discussion.

7.4 Experiments

To evaluate the performance of the proposed framework, extensive experiments on benchmark datasets have been conducted by measuring the throughput (number of operations processed per second) of concurrent operations. The design of experiments is illustrated in Figure 60. The benchmark datasets were generated by a network-based moving objects generator [67] using the road network of the City of Oldenburg, as shown in Figure 61. Three classes of the 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 one-dimensional space, Hilbert curves with different orders were applied in the framework. Based on the moving object set and 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 of the proposed framework.

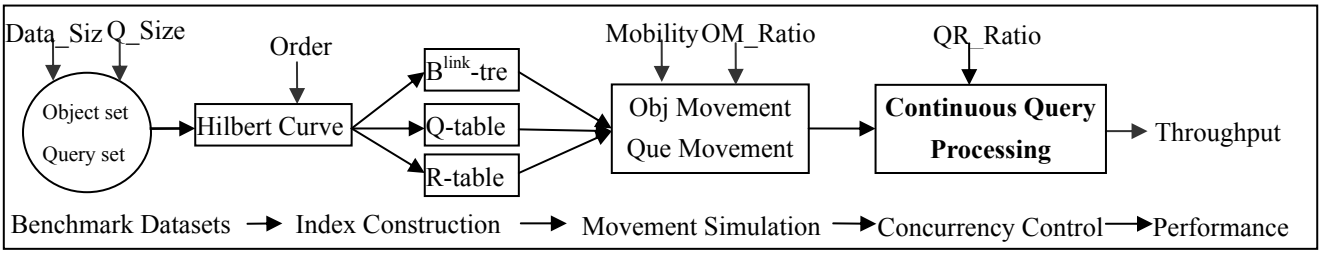


Figure 60. Experiment Flow.

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

- **Order**: the order of Hilbert curve applied. It determines the number of cells for the whole space, and affects the size of the B^{link}-tree and the *Q-table*.
- **Data_size**: the number of initial moving objects/moving queries. It represents the size of the moving object database and the number of continuous queries.
- **Mobility**: the total number of concurrent location updates for objects and queries in a batch. It corresponds to the frequency of object/query location update.
- **OM_ratio**: the percentage of object location updates in Mobility. It reflects the relative update frequency between objects and queries.
- **QR_ratio**: the portion of query reports compared to Mobility. It shows how frequently the query report operation is requested.
- **Q_size**: the side length of query window for each moving query. It simulates query ranges in different applications.

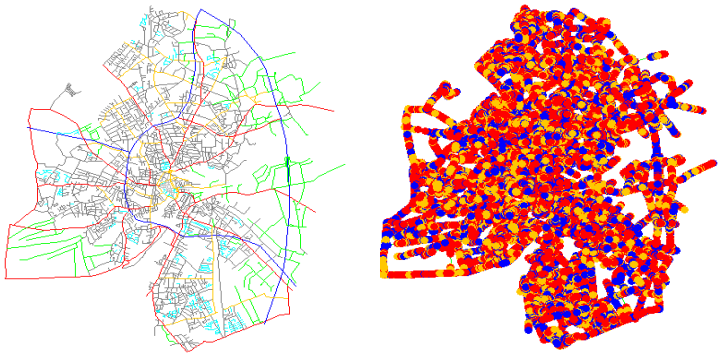


Figure 61. Road Network of Oldenburg and Dataset.

The performance of the framework is evaluated by varing these parameters. The default settings and ranges of these parameters are listed in Table 10.

Table 10. Experiment Parameters Setting.

	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 has been implemented in Java using JDK 1.5. The experiment system was built on a desktop with a Pentium-D 2.8 GHz CPU, and 2GB memory. 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 throughput under different parameter settings was calculated by collecting the average processing time of ten batch executions. For comparison, the continuous query processing extended from CLAM [72] 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 the B^{link}-tree nodes, and release them at commit points. 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 throughput than the pure lock-coupling approach. Note that the proposed approach was compared to a sophisticated method to demonstrate its advantages. Since there are no existing protocols to provide serializable isolation for continuous query processing, to the best of our knowledge, CI is the most proper solution to compare with. The detailed experiment results are presented in the following sections.

7.4.1 Throughput vs. Mobility

In this set of experiments, the impact of mobility was studied by capturing the throughput 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 62, 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 throughput 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 that of 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.

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.

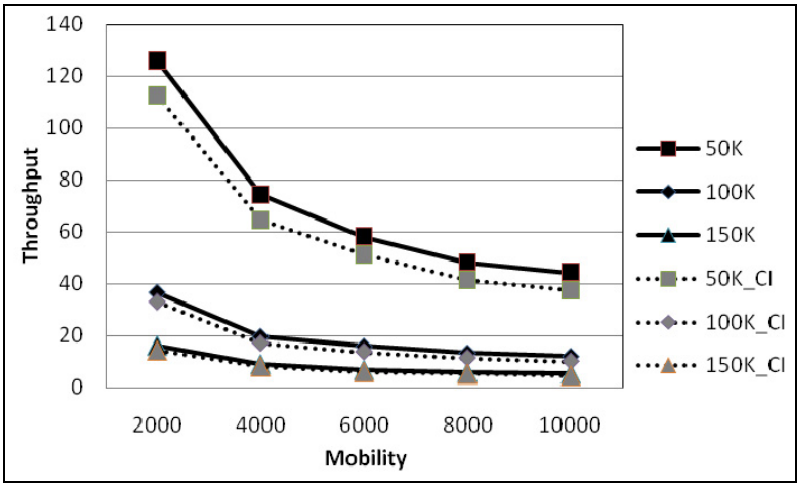


Figure 62. Throughput vs. Mobility.

7.4.2 Throughput vs. OM_ratio

This set of experiments demonstrated the trend of throughput when increasing the OM_ratio. A higher OM_ratio means more object movements within a given mobility. The results are shown in Figure 63, 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 the OM_ratio increased from 10% to 90%.

When the OM_ratio was set to 0.1, the system performed well on all of the three data sets. These high throughput 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.

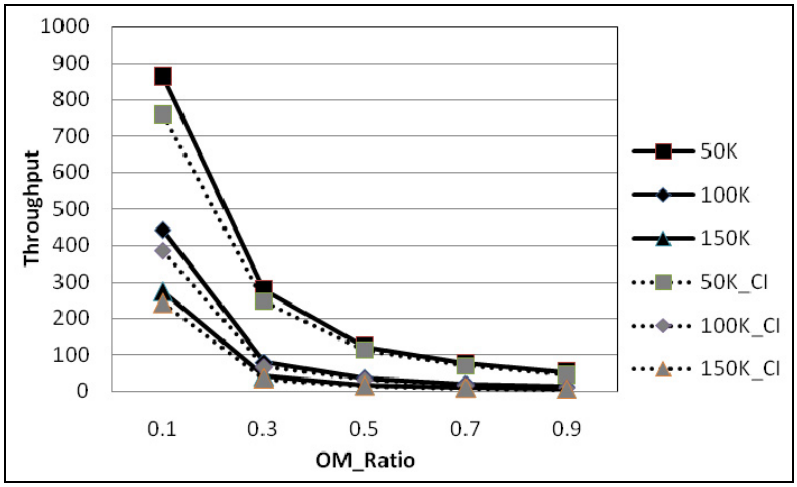


Figure 63. Throughput vs. OM_ratio.

The corresponding throughput of the CI approach showed a similar trend, but was always lower than the proposed framework by 10~20%. The improvement was more significant when the 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 improved as expected.

7.4.3 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 64, where the X-axis represents Q_size and the Y-axis shows the throughput. In all of the three data sets, although the smaller data sets outperformed the larger ones, the throughput 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 compared to the entire 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.

Similarly, the throughput of the data sets applying the CI approach was 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.

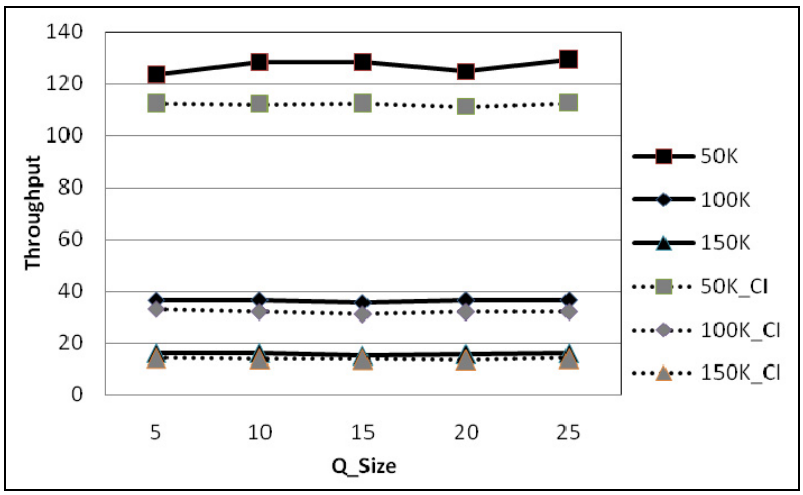


Figure 64. Throughput vs. Q_size.

7.4.4 Throughput vs. QR_ratio

The focus of this set of experiments was to study the impact of the QR_ratio on concurrent continuous query processing. The QR_ratio was gradually increased from 5% to 25%, and the corresponding throughput on three data sets was collected to be compared. The results are plotted in Figure 65, where the X-axis indicates the QR_ratio and throughput is represented in the Y-axis.

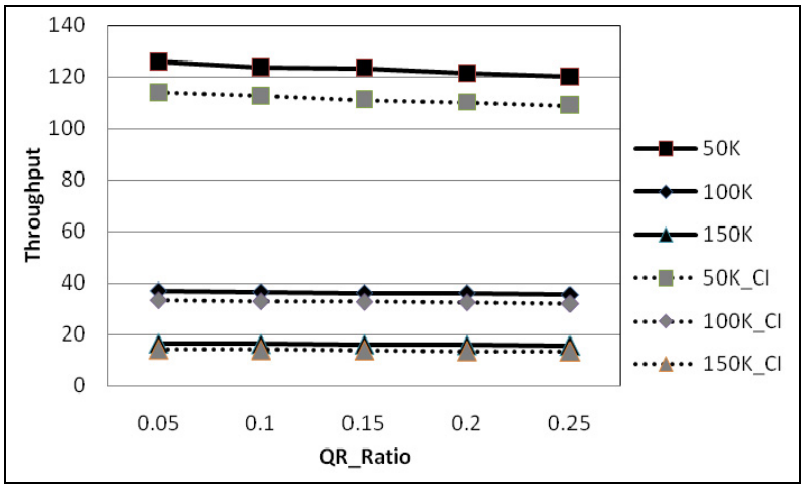


Figure 65. Throughput vs. QR_ratio.

As shown in the figure, the throughput of three data sets slightly decreased when the QR_ratio increased significantly. When the QR_ratio increased from 5% to 25%, the throughput only reduced about 5%. These

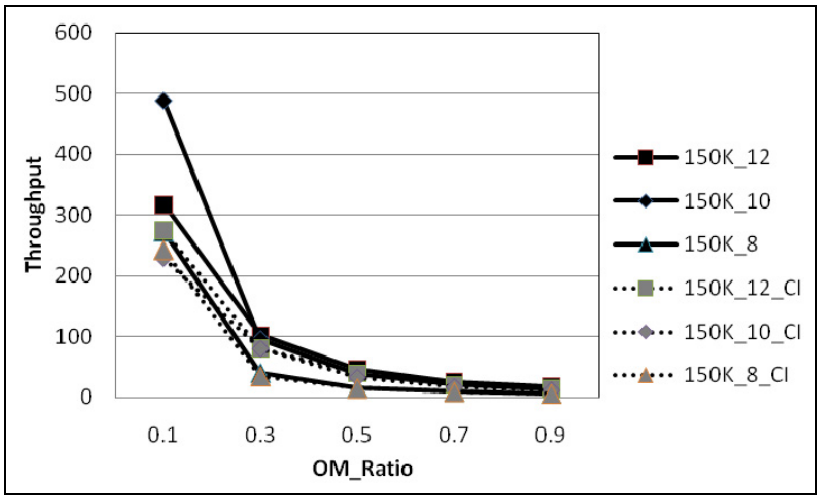
results suggest that the cost for a query report operation is negligible compared to object movement and query movement. These results can be well explained via the design of this concurrent continuous query framework. Query report, as illustrated in Algorithm 17, only requests a read-lock on the given query, and reads the corresponding entry in the R-table. It is a memory-based operation and can be quickly processed to avoid blocking other operations.

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

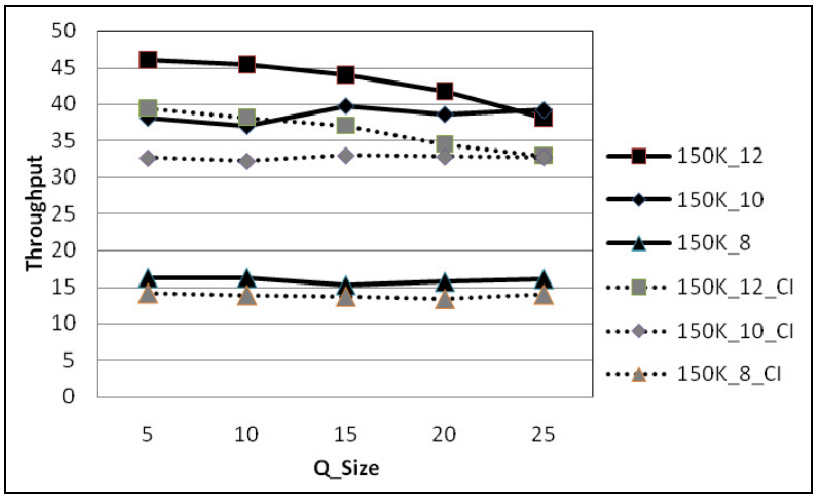
7.4.5 Throughput vs. Order

This set of experiments demonstrates the impact of the order of SFC mapping on the performance of concurrent continuous query processing. 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 *ID*s will be indexed in the B^{link}-tree. In other words, less objects will be contained in a single cell.

The OM_ratio and the Q_size were varied in the experiments to investigate the impact of order in different scenarios. As illustrated in Figure 66 a), when the OM_ratio increased from 0.1 to 0.9, the throughput of the system on different data sets kept decreasing. The reason for that has been discussed in Section 7.4.2. 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 with order 8. That is because: 1) When SFC order is higher, the B^{link}-tree has more nodes, and the locks on the B^{link}-tree nodes have less chance 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 the 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.



a) Over OM_ratio.



b) Over Q_size.

Figure 66. Throughput vs. SFC Order.

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 significantly as the proposed approach, because of its increased lock durations caused by Hilbert mapping calculation.

When the Q_size was increased from 5 to 25 in Figure 66 b), the Hilbert curve with order 12 performed better than with 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 with regard to the Q_size,

behaving similarly as discussed in Section 7.4.3. However, the Hilbert curve with order 12 exhibited a 20% performance drop in the figure when the Q_size increased. These results suggest that the cell size in the Hilbert curve with order 12 is small enough to be comparable with the query sizes. Therefore, a query may need to scan multiple cells to locate the moving objects.

According to the above mentioned experimental results, the proposed concurrent continuous query processing optimizes the locking strategy and improves the concurrency level. The parameters, including order, mobility, data_size, and the OM_ratio are found to have significant impact on the performance of the proposed framework. Within these parameters, the increasing of mobility, data_size, or the OM_ratio will degrade the system performance, whereas a higher order may promote the throughput.

7.5 Conclusion

This chapter 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-freedom. Its correctness has been proved by analyzing the lock durations of the operations, and the performance has been evaluated by a set of experiments on benchmark datasets. 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.

Chapter 8. DIME: DISPOSABLE INDEX FOR MOVING OBJECTS

This chapter proposes a generic index framework, DIME: Disposable Index for Moving objEcts, for efficient moving object management. The proposed disposable index eliminates delete operations on the indexing structure, provides parallel query ability, and reduces overhead for concurrency control. Both snapshot and continuous query processing has been designed for this index. Experimental results on benchmark datasets demonstrated the scalability and efficiency of the proposed disposable index.

8.1 Preliminaries

In moving object management, each location update needs to update the index twice, for inserting the new location and deleting the old location. Significant effort is required to remove the obsolete information and thus impairs the performance. As shown in the literature [17], the update I/O cost could be improved 44~68% by caching delete operations. However, these cached delete operations still need to revise the index structure at certain time points. Further improvement can be expected if delete operations no longer change the index tree. On the other hand, existing moving object index approaches do not natively support parallel computing structures. Thus additional management (e.g., slice or copy) is needed to implement these indices in modern parallel structures. One location update will need to modify multiple slices or copies of the index, which increases the risks of inconsistency and the complexity of the database system.

We propose an index framework, DIME, to eliminate delete operations on the index structure and provide parallel query ability. Since the locations of moving objects are updated frequently, location information can become obsolete within a given period of time. For this scenario, we provide a solution to conserve 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 removed without changing the internal structure. In the proposed solution, old locations of moving objects are disposed from the index without index modification. To address the parallel processing issue, we design a moving object index structure that is natively sliced based on the update timestamps of moving objects.

Before presenting the construction of a disposable index and the corresponding query processing algorithms, we introduce the overall design of the proposed framework in this section.

8.1.1 Terms and Assumptions

In this framework, as illustrated in Figure 67, a new component of spatial index is constructed every time 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 included in each Δt_{mn}. Each index component is an independent spatial index for the locations updated in corresponding period Δt, with lifetime (n+1)*Δt. During its lifetime, an index component can be traversed to answer spatial queries, but does not respond to any delete operations. According to the definition of Δt_{mn}, after has been initiated for (n+1) phases, a component only contains obsolete locations. Therefore this component is entirely disposed. With a proper phase number, each index component should have no overlap with its predecessor and successor. The above concepts are summarized in Table 11.

Table 11. Terms.

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 index component
Lifetime	Lt = (n+1)*Δt	Time period from constructing an index component to disposing it

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 query is represented as a spatial box, which is the query window; each query submits its query window to the database once.
- **Continuous query**: Each continuous query is represented as a moving query window; each query periodically reports its new query window to the database.

These assumptions are applicable in many real-world applications.

8.1.2 Framework

Based on the above assumptions, a moving object access framework based on the DIME is designed as follows. This framework consists of a set of index components for snapshot query processing, and two hash tables in addition for continuous query processing, as shown in Figure 67. The supported spatial operations

are location update, window search, object movement, and query movement. A **location update**, as illustrated in the figure, updates the most current index component ($T+n\Delta t$) by inserting the new location. It then logs its deletion to the index component that contains its old location (T). A **window search** performs window queries on each existing index component, validates the results against deletion logs, and retrieves the validated results covered by the search window at the query time. An **object/query movement** (occurs in continuous query processing) performs a location update on the object/query index, searches the query/object index, and updates the query results. As shown in the figure, the continuous queries are indexed in the *Q-table*, and the continuous query results are indexed in the *R-table*.

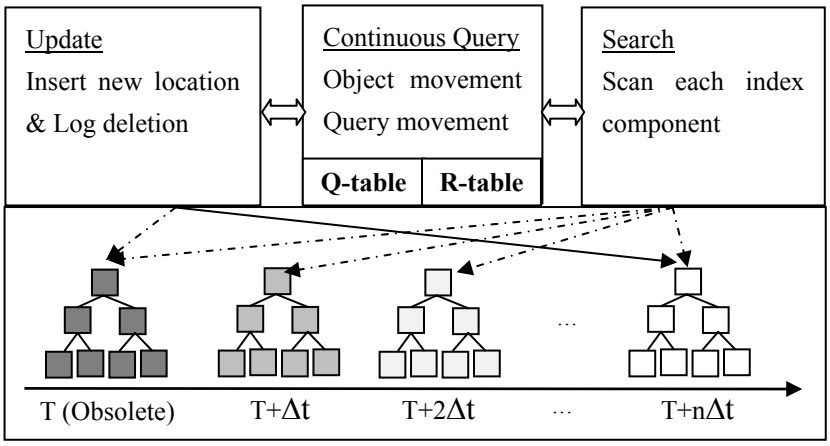


Figure 67. Framework of DIME.

Disposable index is a generic framework, as any spatial index structure can be applied to construct the index components. To better demonstrate the design and algorithms, we apply a linear spatial index (B+-tree integrated with SFC) in the rest of this paper. The Hilbert Space-Filling curve [3], 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 spatial locations of objects can be represented by one-dimensional cell *ID*s. The cell *ID*s of moving objects can 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, two additional hash tables (*Q-table* and *R-table*) are applied to index current locations of queries and store the query results. The cell *ID*s of moving queries are indexed using a hash table, *Q-table*, where the cell *ID*s 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-tree, is that point query is the only search on

the query index in the proposed algorithms.

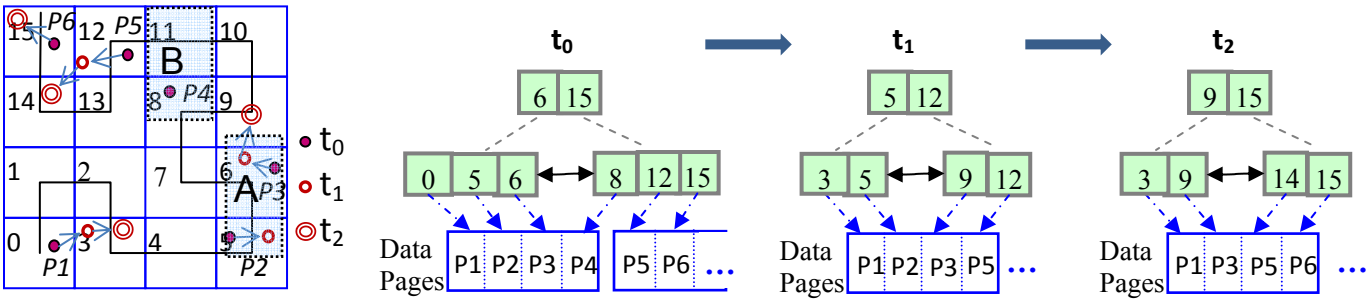


Figure 68. An Example of Moving Objects and Corresponding B+-trees at t0, t1, and t2.

8.1.3 Illustration

An example of moving object dataset is illustrated in Figure 68, where three timestamps, t_0, t_1, and t_2 are included. Note that $t_1 = t_0 + \Delta t$, and $t_2 = t_1 + \Delta t$. The whole 2-dimensional space is divided into equal-sized square cells. These cells are associated with 1-dimensional *ID*s using a Hilbert curve [3] with order 3. The B+-tree for t_0 shows the initial index component for this dataset. In this initial component, all the existing moving objects at timestamp t_0 are indexed according to their SFC cell *ID*s. Any object that updates its location between t_0 and t_1 will have the new location indexed in the index component for t_1, as shown in Figure 68. Similarly, the B+-tree for t_2 shows the index component for all the moving objects that update locations between t_1 and t_2. In case n equals to 2, the index components for t_1 and t_2 can cover all the moving objects.

On t_2, when the index component for t_2 is completely constructed, the index component for t_0 can be disposed. Therefore, at any time, a disposable index contains n constructed index component and one constructing index component.

Update - In this index framework, each index component does not change after being completely constructed. Any query movement calculates the new location of the query based on the timestamp of the constructing index component. Then it inserts that new location into the constructing component, and marks the deletion of its old location in a constructed component. For instance, in Figure 68, the object *P1* 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 located in cell *3*. Then l_{t1} is inserted into the index tree for t_1 (as shown in the B+-tree for t_1), and a note is created to flag its obsolete record in the index tree for t_0. Similarly, the object *P2* in cell 5 in Figure 68 updates it location by t_1. This object is then indexed by the B+-tree for t_1. Since it does not report a new location after t_1, the B+-tree for t_2 does not contain this object.

Similarly, the object *P4* in cell 8 does not update its location since t_0, so it is only indexed in the B+-tree for t_0. If it does not update after the maximum time interval Δt_{mn}, the system will need to estimate its new location and insert into the current constructing B-tree.

Search - Since the whole set of moving objects are covered by *n* index components; a window query needs to traverse all these components to retrieve the qualified objects. In these traversals, the query windows are revised according to the object velocities and component timestamps. For example, a window query issued at t_2 needs to traverse both the index components for t_1 and t_2. Its original query window is applied on the component for t_2. A revised window, enlarged with the maximum object velocity at t_1, is applied on the component of t_1. The results from the component of t_1 are then validated against the obsolete flags to remove obsolete locations, and combine with the results from the component for t_2 as the final result set.

Q-table & R-table - An example of a *Q-table* for query index is shown in Figure 69. 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*. 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 *ID*s are used as hash keys, and each entry stores a list of objects covered by a particular query. Figure 69 shows an example of an *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. Because the update cost on hash tables is very low compared to the B-tree, it is not necessary to apply the design of disposable index on the *Q-table* and the *R-table*. However, the query evaluation needs to consider all the existing index components of moving objects for a complete set of results.

Q-table

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

R-table

Query:	A	B
Object:	P2, P3	P4

Figure 69. Q-table for Queries and R-table for Query Results.

In this indexing framework, an object movement will update its location in the disposable indexing structure, then search the *Q-table* for affected queries, and finally refresh the corresponding query results in the *R-table*. For example, if the object *P3* in the cell 6 is moving to the cell 9, where both cells overlap with the query *A*, it first inserts its new location in the current B+-tree by adding new entry 9, and logs it deleting on an corresponding old B+-tree. Then the *Q-table* is searched and *A* is found to cover only the old location of *P3*. At last, the record *P3* is removed from the *R-table* under the key *A*. A query movement needs to search the B+-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 *B* is moving towards west by one cell to cover cells 12 and 13. The system searches the B+-tree using cells 12 and 13, validates the results, and identifies the object in cell 12. Then the *Q-table* is updated by removing entries 8 and 11, and creating two new entries 12 and 13 with *B* inside. The *R-table* is refreshed by inserting *P5* into and deleting *P4* from the entry *B*. A query report operation simply visits the entry in the *R-table* and outputs its object list.

Similarly, a disposable index can be constructed by utilizing the R-trees as index components. An example is shown in Figure 70, where the R-trees for three phases, t_0, t_1, and t_2 are created for moving objects. Similarly to the example in Figure 68, the disposable index with R-trees constructs one R-tree for each phase to track the newly updated locations. In other words, the R-tree for t_1 only indexes the positions updated between t_0 and t_1, and the R-tree for t_2 only indexes the locations updated between t_1 and t_2.

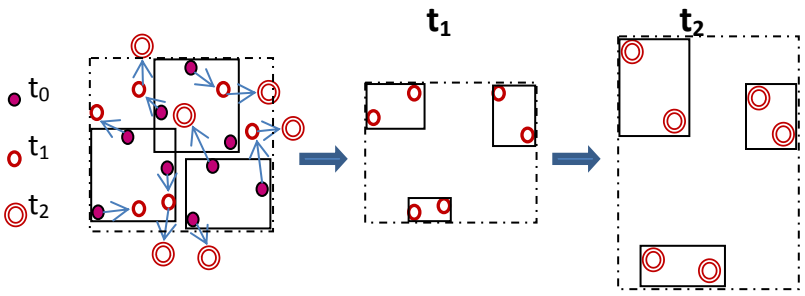


Figure 70. Disposable Index Constructed with the R-trees.

8.2 Disposable Index Operations

Operations on disposable index cover both snapshot queries and continuous queries. Snapshot query processing introduces location update and window search; continuous query processing requires object movement, query movement, and query report. These operations are designed as follows.

8.2.1 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.

8.2.1.1 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 structure includes a B+-tree forest as the disposable index, and a log buffer associated with each B+-tree. This operation inserts the new location into the current index component, and reports the deletion of the old location to the index component corresponding to the last update time. Algorithm 18 presents the two phases of location update on disposable index, insertion and deletion.

```
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 (A Forest of
    B-trees)
Output: Nil

//Phase1: Insertion
loc_cur = (DI[now].TS – TS_new) * vel_new + loc_new;
c_new = SFC_map(loc_cur); //determine the cell contains loc_cur
n_new = DI[now].traverse(c_new); // locate the leaf which contains c_new
n_new.insert(c_new); //update B-tree if necessary for inserting loc_cur

//Phase2: Deletion
DI[TO] = DI.find_tree(TS_old); //find the B-tree for object's last update
If (DI[TO]!=DI[now])
    DI[TO].mark_obsolete(Oid); //mark the deletion of Oid
Else
    DI[now].mark_obsolete(Oid, loc_cur); //mark the valid location

Return;
```

Algorithm 18. 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 Tree's Timestamp – reported Timestamp) * reported Velocity + reported Location. Thus, all the moving objects that are indexed in one B+-tree have normalized locations based on the timestamp of the tree. The selected SFC is then

utilized to assign the spatial location a one-dimensional cell *ID*. Using this cell *ID*, the new location is inserted into the current B+-tree.

Phase 2, Deletion. The algorithm identifies the B+-tree that indexed the previous location of this object, according to its last report timestamp. After that, a log is added to that tree to indicate the deletion of this object from that B+-tree. In case the previous report timestamp corresponds to the current B+-tree, a log will be added to state the object's valid location on the current B+-tree. This only happens on objects that update more than once in a phase. Note that with these logs, the deletion on that B+-tree is not required.

In the example in Figure 68, 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 the location of this object in the B+-tree for t_0 is obsolete.

8.2.1.2 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 within the range at the search time. This operation traverses each existing B+-tree in the disposable index to retrieve the objects, validates these objects, and returns them as results. The detailed process is described in Algorithm 19.

Phase 1, Parallel Search. In this phase, the algorithm handles the window search on each B+-tree in the disposable index. For each B+-tree, the search window needs to be adjusted using the maximum velocity times the difference between the search time and index timestamp. After retrieving the objects using revised windows, for each B+-tree, the algorithm refines the object set by computing their locations using the individual velocity. Then the object set for each B+-tree is compared against the obsolete logs to remove invalid objects. This phase can be executed on a parallel computing hardware to obtain premium performance.

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

For instance, assuming Δt_{mn} contains two phases, 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.

Algorithm Window_Search
Input: TS_Q: Query Timestamp, Win: Query Window, DI: Disposable Index of Objects (A Forest of B-trees)
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]*
c_que[i] = SFC_map*(Win[i])*; //determine the cells overlap with *Win[i]*
Res[i] = *DI[i]*.search*(c_que[i])*; //retrieve the objects covered by *c_que[i]*
For (int *j* = 0; *j* < *Res[i].size*; *j*++)
If (*Res[i]*.get*(i).current_location* Not_In *Win*)
Res[i].remove *(i)*; //refine the results
Res[i] = *Res[i]* – *DI[i].obsolete_set*; // remove the objects marked as obsolete in *DI[i]*

//**Phase2: Union**

$$Res = \bigcap_i Res[i] \bigcap_i Res[i] \bigcap_i Res[i];$$

Algorithm 19. Window_Search.

8.2.2 Continuous Query Processing

Taking the snapshot query processing as the fundamental operations, continuous query processing can be designed by utilizing the Q-table and R-table.

8.2.2.1 Object Movement

The operation of object movement takes the object *ID*, last update time, current update time, new velocity, and new location of the moving object, as well as the disposable index, *Q-table* and *R-table*, as input parameters. This operation updates the object location on the B+-tree forest, 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 20.

Phase 1, **Object Location Update.** This operation performs a location update (Algorithm 18) for the object. It first applies the SFC to find the cells that cover the new location of the object. After the cell *ID* is determined, the algorithm traverses the current B+-tree to locate the leaf nodes that contain this cell *ID*. Meanwhile, a log for deletion is added to the index 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-table*. Then the *Q-table* is accessed to retrieve the candidate queries linked to these cell *ID*s. A list of affected queries is retrieved by computing their topological relation with the exact old location and new location.

```
Algorithm Object_Movement
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 (A Forest of B-trees), Q: Q-table, R:
    R-table
Output: Nil

//Phase 1. Object location update
Location_Update(Oid, TS_old, TS_new, loc_new, vel_new, DI);

//Phase 2. Query search
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

//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

Return;
```

Algorithm 20. Object_Movement.

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 from, and is added into the entries for the queries that the object is moving to.

Take the object movement example in Figure 68, and assume at t_2, the object *P3* in the cell 6 is moving into the cell 9 and out from the query *A*. This algorithm first locates the leaf nodes that contain the cell 9 or will contain the cell 9 if it does not exist yet. A log of deleting *P3* is recorded for the B+-tree for t_1, and a new entry for the cell 9 is inserted into the B+-tree for t_2. In Phase 2, the algorithm queries the *Q-table* for the queries cover cell 6 and cell 9. It retrieves the query *A* for both cells, and determines that the query *A* covers only the old location of *P3*. In the last phase, because *P3* is moving out of the range of query *A*, it is removed from the entry of *A* in the *R-table*.

8.2.2.2 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 16 shows the details of the query movement operation.

Phase 1, **Object Window Query**. The algorithm applies the SFC to locate the cells overlapped by the new query window. The B+-tree is then traversed to retrieve all the objects covered by the new query window, utilizing the window_search algorithm (Algorithm 19). A set of objects is retrieved at the end of this phase.

Phase 2, **Query Location Update**. The algorithm updates the corresponding entries of the *Q-table* by adding or removing the query *ID*.

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.

For instance, the following steps execute the query movement based on the object and query example in Figure 68. Assume the query *B* moving towards west by one cell to cover cells 12 and 13 at t_1. In Phase 1, all the existing B+-trees (the B+-trees for t_0 and t_1) are searched using a modified range of query *B*. In this

example, the object *P5* is returned as the result set. In Phase 2, the *Q-table* entries for cells 8 and 11 remove the query *B*, meanwhile the entries for cells 12 and 13 add *B*. In Phase 3, the object *P5* is inserted into the entry *B* in the *R-table*. After these three phases, a query report operation of *B* can retrieve object *P5* as the up-to-date query result.

Algorithm Query_Movement
Input: TS_Q: Query Timestamp, Win: Query Window, DI: Disposable Index of Objects (A Forest of B-trees), Q: Q-table, R: R-table
Output: Nil

//Phase 1. Object window search
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*
o_new = Window_Search*(TS_Q, Win, DI)*; // find the objects overlapping with *win_new*

//Phase 2. Query location update
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*; //add *Qid* into *Q-table*
PageUpdate(*Qid, win_new*); //update query window

//Phase 3. Result refresh
R.entry(*Qid*).objList = *o_new*; //update *R-table* entry

Return;

Algorithm 21. Query_Movement.

8.3 Performance

The proposed disposable index reduces the location update cost by eliminating the index modification for delete operations. A popular spatio-temporal index, the B^x-tree, constructing a new subtree for each phase, has demonstrated outstanding performance for moving object management. Therefore, we analyze the performance of the disposable index by comparing against the B^x-tree, in terms of I/O cost and space cost.

8.3.1 I/O Cost

Assuming the cost of reading a data page is *CR*, writing a data page is *CW*, traversing a B+-tree is *CT*, and modifying a B+-tree is *CM*, a typical delete operation costs *CR+CW+CT+CM* on a B^x-tree. Since the disposable index does not require modifying the B+-tree for delete operations, it only needs to log the deleted object *ID* in memory, whose cost can be neglected comparing to the disk-based I/O on a B^x-tree. In order to dispose the obsolete index components, the disposable index requires one I/O to mark the deletion of that B+-tree file and another disk I/O to delete the corresponding data file. Suppose there are *k* objects indexed in one index component in average, the I/O cost for one deletion on a disposable index becomes 2/*k*CM*. The overall location update cost on a B^x-tree, including an insertion and a deletion, can be calculated using the following expression: 2*(*CR+CW+CT+CM*). On the other hand, the location update cost on the disposable index is only *CR+CW+CT+(k+2)/k*CM*, which is half of that cost on a B^x-tree when *k*>>2.

The costs for a search operation on both B^x-tree and disposable index are identical, because they use similar index structures. Although the disposable index 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 both indices can be represented as *CT + CR*l*, assuming *l* data pages are retrieved on average.

8.3.2 Space Cost

Similarly to the B^x-tree, the disposable index contains multiple components that correspond to different time periods. Because the disposable index 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 disposable index and the B^x-tree, and *k* objects update their locations in each phase. Because the size of a B-tree is proportional to the number of keys, it can be denoted as *p*k*. Utilizing the above notations, the size of the disposable index can be calculated. According to the design of the disposable index, there are *n+1* B+-trees at the same time. In those trees, *n* are fully constructed and one is being constructed. Therefore, the total size of this index structure is between *n*p*k* and *(n+1)*p*k*.

In the B^x-tree, the number of indexed objects is constant if the object set is fixed. There are *n+1* sub-trees in a B^x-tree at the same time, and each sub-tree has different size. For easy calculation, we assume the object

update time intervals are evenly distributed between Δt and $n*\Delta t$. Therefore, in any phase, k/n objects are removed from each sub-tree. The total size of this B^x-tree is $(0+1/n+2/n+...+1)*p*k = (n+1)*p*k/2$.

In the disposable index, a main memory buffer is used to log the obsolete objects. Since only the object *ID*s are stored in this buffer, the memory cost for logging the obsolete objects is *size_of(object ID) * (# of objects in disposable index - # of objects in the B^x-tree)*. Thus, the upper bound of the buffer size is *size_of(object ID) * (n+1) * k/2*. In case each object *ID* is a 4-byte integer, *k* is 1 million, and *n* equals to 4, the maximum buffer required in the disposable index is 10 MB. In other words, given a 10 MB buffer, the disposable index can handle over 1 million location updates in each phase.

8.3.3 Parallel and Concurrency

The structure of the disposable index allows parallel search on the index components. Distribute each index component to an independent storage or even a computer node. Then a search operation can traverse each index in parallel for optimal performance. The result set returned from each component can be merged together in a pair-wise manner. Suppose the query processing time on one index component is *T*, the total processing time for a parallel search on a disposable index with n phases will become $T+\lceil \log(n+1) \rceil * TM$, where *TM* indicates the time cost for one merge.

From the aspect of concurrency control, less overhead is required on this index framework. Because only the current index component will be modified, other live components do not need to handle read-write conflicts. The current index component only contains a subset of the dataset, therefore, compared to indexing the whole dataset, DIME needs to maintain less locks.

In summary, the proposed disposable index requires half of the update cost and similar search cost compared to the B^x-tree. On the other hand, it takes as much as 200% of the index storage of the B^x-tree, with an appropriate buffer size. Furthermore, the query performance on the disposable index can better utilize the advantage of the parallel search.

8.4 Experiments

To evaluate the performance of the proposed framework, a set of extensive experiments on benchmark data sets has been conducted by measuring the I/O cost (average number of I/O consumed) of operations on moving objects. The design of experiments is illustrated in Figure 72. The benchmark data sets were

generated by a network-based moving objects generator [67] using the road network of the City of Oldenburg, as shown in Figure 71. 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 a one-dimensional space, the Hilbert curves with different orders were applied in the framework. Based on the moving object set and the Hilbert curve, an initial B+-tree was constructed. On the other hand, the object movements simulated by the generator were translated into object location updates (and query updates for continuous query processing). The overall page accesses for each set of operations was collected to calculate the I/O costs.

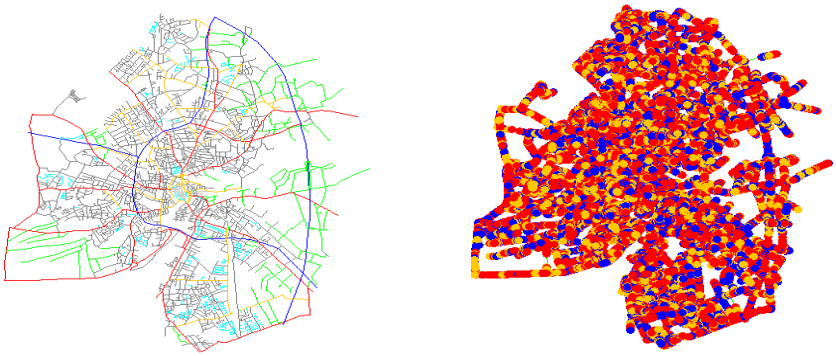


Figure 71. Road Network of Oldenburg and Data.

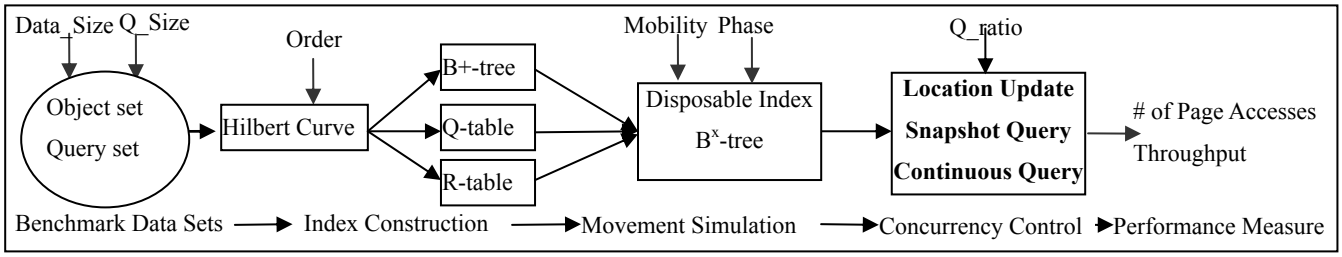


Figure 72. Experiment Design.

In the experiments, six parameters were varied to simulate different application scenarios and demonstrate their impact on 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. It determines the size of the initial index component.
- **Q_size**: the side length of query window for each query. It simulates query ranges in different applications.
- **Phase**: number of index components constructed in Δt.
- **Mobility**: the number of location updates for objects issued in a phase.

- **Q_ratio**: the number of queries (query location updates for continuous query) compared to Mobility.

The performance of the framework is evaluated by varying these parameters. The default settings and ranges of these parameters are listed in Table 12.

Table 12. Experiment Parameters.

	Order	Data_Size	Mobility	Phase	Q_ratio	Q_size
Default	8	300K	100K	2	25%	100
Range	8~10	100K~300K	25K~150K	2~7	5%~30%	100~1000

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 and 2 GB memory. The I/O cost per operation under different parameter settings were calculated by collecting the average number of page accesses of ten rounds of executions. To evaluate the effectiveness of the parallel sructure of disposable index, the throughput of the search operations was counted for both sequential and concurrent executions. Three sets of initial moving objects and moving queries were used in the experiments, with data_size 100K, 200K, and 300K, respectively. The results showed that the initial size of the dataset did not affect the operation performance, because the initial index component was disposed after $n+1$ phases. Thus, the results from the 300K dataset are used for the following evaluation.

For comparison, the query processing based on the B^x-tree was implemented. The B^x-tree is known as one of the most popular moving object indices to provide timely query results efficiently. The B^x-tree generates a new subtree in each phase, processes location updates by inserting new locations and deleting old locations on different subtrees. The search operation on the B^x-tree traverses each subtree to find the objects. We use Bx to represent the B^x-tree approach, and DI for Disposable Index, with the followed number indicating the order of the Hilbert Curve. A Hilbert Curve with a higher order generates finer grids in the data space, and usually results in a B+-tree with more nodes. The detailed experimental results are presented in the following subsections.

8.4.1 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.

8.4.1.1 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 continuous query processing with different numbers of movements. The experiment results are illustrated in Figure 73, 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 index 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. This can be verified in Figure 73, when the mobility increased from 25,000 to 150,000, 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 DI and Bx increased along with the mobility. Especially when the mobility changed from 25,000 to 100,000, the average number 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 25,000 to 50,000. When the mobility was 50,000 or 75,000, the indexing trees were relatively sparse, therefore structure changes easily occurred.

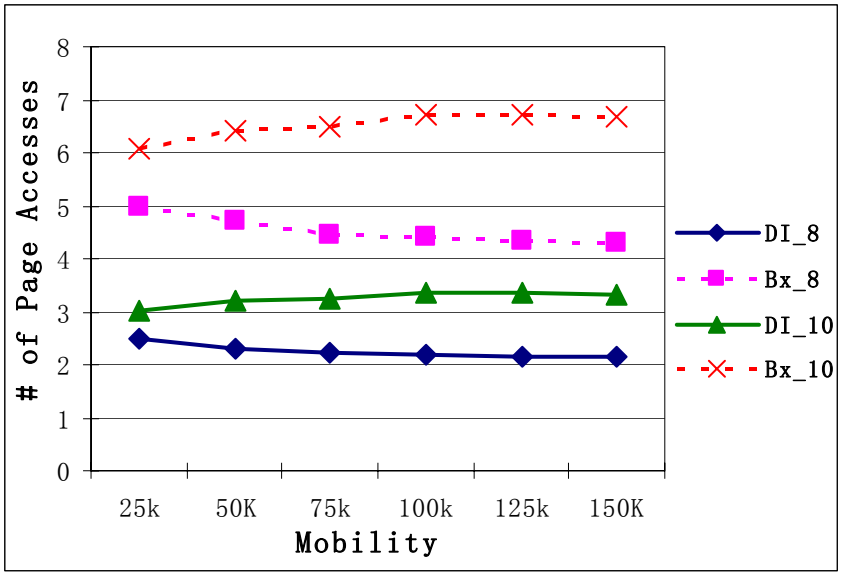


Figure 73. Update I/O vs. Mobility.

Comparing the average number of page accesses between the disposable index and the B^x-tree, it is clear that the B^x-tree consumed twice I/O compared to the disposable index, for both order 8 and order 10 SFCs. This result is consistent with the theoretical analysis in Section 8.3. That is because that one update in the

disposable index only needs to perform insertion, while an update requires insertion and deletion in the B^x-tree. From this set of experiments, we can conclude that the update I/O of the disposable index is half of the B^x-tree.

8.4.1.2 Search I/O vs. Q_size

Performance of search operations against query size was studied in this set of experiments, as shown in Figure 74. In this figure, 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 the figure, 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 130. The I/O cost of order 10 increased faster than that of order 8 in the figure, because the B+-trees with order 10 contains more nodes, and the same area can cover more nodes in those trees than the B+-trees with order 8.

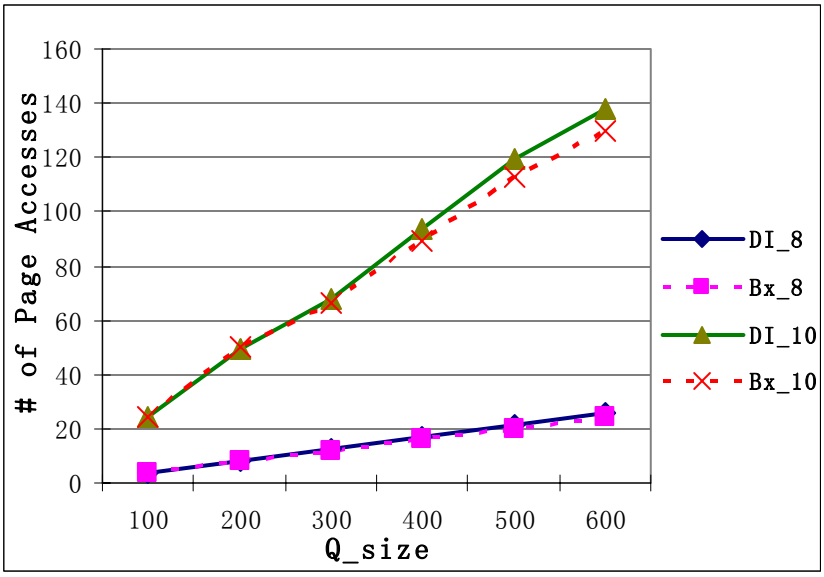


Figure 74. Search I/O vs. Q_size.

When the Q_size exceeded 400, the performance of DI was slightly lower than that of Bx. 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 indepent B+-trees in the disposable index. Observed from this set of experiments, the B^x-tree and the disposable index performed similarly on search cost, and the B^x-tree slightly outperformed the disposable index on large query windows.

8.4.1.3 Search I/O vs. Phase

The number of phases determines the number of B+-trees that exist in the disposable index, and the number of subtrees in the B^x-tree. 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 Figure 75, where the *X*-axis indicates the number of phases and the *Y*-axis represents the average number of page acesses in each search operation. According to the search algorithm, all the index components need to be traversed to get a complete result set. In this figure, both DI and Bx 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 87.

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

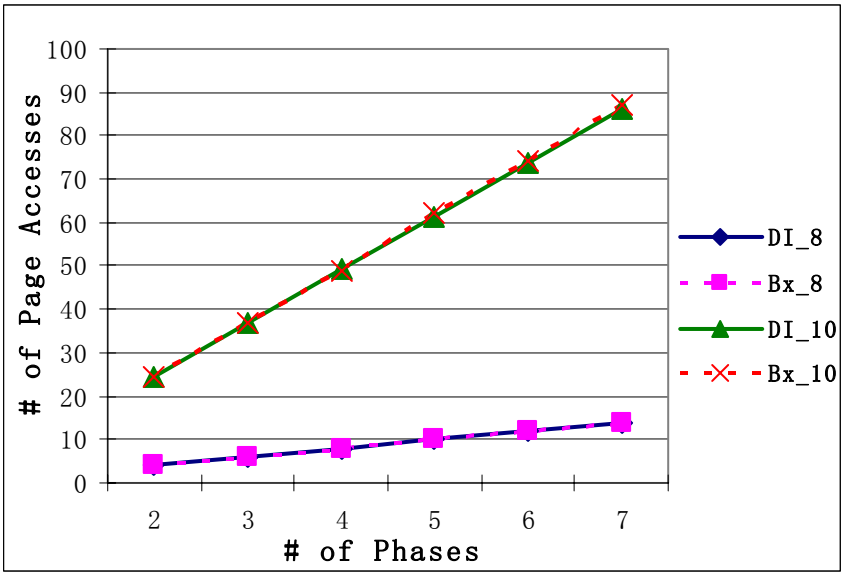


Figure 75. Search I/O vs. Phase.

8.4.1.4 Search I/O vs. Mobility

The mobility determines the size of each index component. This set of experiments analyzes the impact of mobility on search cost. The results are shown in Figure 76, 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 25,000 to 150,000 in the experiments. When the order was 8, there was no significant change observed for DI and Bx as the mobility increased. Both DI_8 and Bx_8 spent about 4 page accesses regardless of the change of the mobility. When the order of SFC was 10, a significant jump occurred between 25,000 and 50,000. The average number of page accesses increased from 17 to 24 for both DI and Bx, because the height of each index component was increased when the mobility reached 50,000. After this jump, both DI and Bx accessed about 24 pages on average constantly.

Similarly to the previous experiments, the search performance of DI and Bx was very close. This observation matches the performance analysis in Section 8.3.

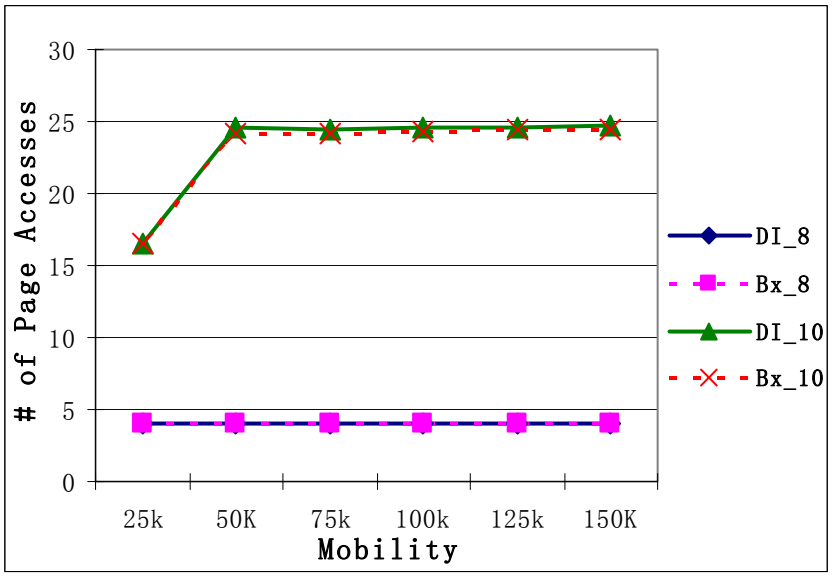


Figure 76. Search I/O vs. Mobility.

8.4.1.5 Search Throughput vs. Phase

The structure of the disposable index is suitable for parallel search. To estimate the performance gain from this design, the throughput of search operations was collected for both sequential execution and concurrent execution. Figure 77 shows the results of the comparison of the throughtput, where the *X*-axis

represents the number of phases, and the *Y*-axis indicates the throughput. When there was only one phase, similar performance was achieved by DI_Exp_10, DI_Exp_5, DI_Serial, DI_Concur, and Bx. When the number of phases increased, the Bx and DI_Serial decreased their performance with similar rates. However, the concurrent execution decreased its throughput much slower than the Bx and DI_Serial. With 7 phases, the throughtput of DI_Concur was 40, more than twice of the throughput of DI_Serial and Bx.

By applying independent index components in the disposable index, with concurrent execution, the proposed disposable index demonstrated significant performance improvement. Figure 77 also illustrates the expected throughput on a parallel computing hardware structure as DI_Exp_10 and DI_Exp_5, assuming merging results from two nodes costs 10% and 5% of a search on one B+-tree respectively. Theoretically, the throughput of a parallel search operation on the disposable index with *n* phases should be that of a search on only one phase plus transmitting and merging the result sets, as discussed in Section 8.3.

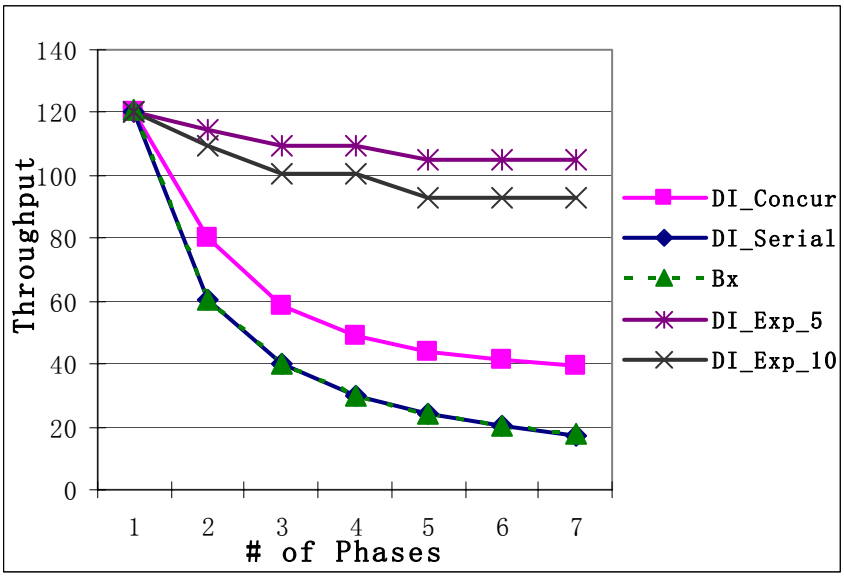


Figure 77. Throughput vs. Phase.

8.4.2 Continuous Query Processing

Continuous query procssing on the disposable index handles object movements and location movements. The number of object movements within one phase is determined by the mobility, and the number of query movements is calculated as *Mobility* * *Q_ratio*.

8.4.2.1 Movement I/O vs. Mobility

Figure 78 demonstrates how the average number of page accesses in a movement changed with regards to the mobility. In this figure, the *X*-axis represents the mobility (from 25,000 to 150,000), 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 this figure are actually the summaries of the curves in Figure 73 and in Figure 76. As shown in the Figure 78, the Bx_8 spent over 70% more page accesses than the DI_8. The DI_10 cost 2.5 less page accesses than the Bx_10 in the experiments.

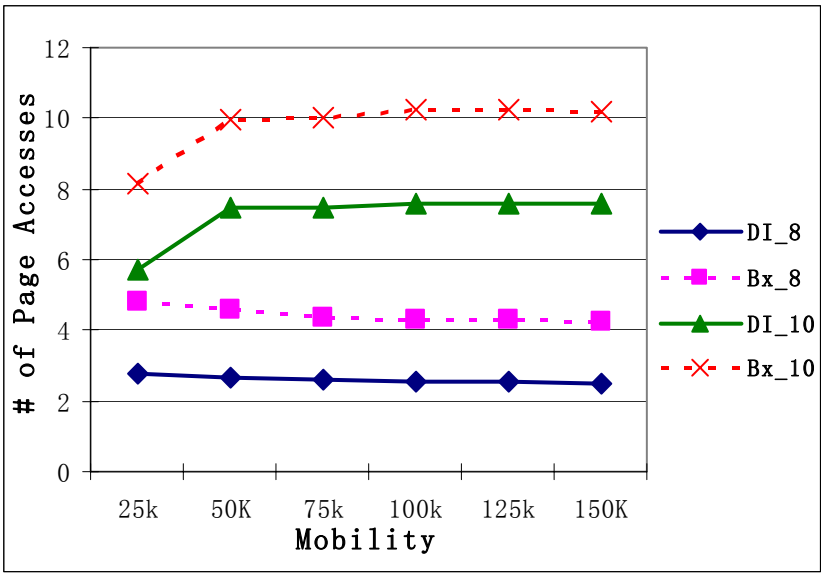


Figure 78. Movement I/O vs. Mobility.

8.4.2.2 Movement I/O vs. Q_ratio

A larger Q_ratio indicates more query movements in one phase. Figure 79 illustrates the impact of Q_ratio on the I/O cost per movement. In this figure, the *X*-axis represents the Q_ratio, and the *Y*-axis indicates the average number of page accesses in a movement (object or query). 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 acesses. With order 8, both DI and Bx performed stably when the Q_ratio changed.

Similar to Figure 78, 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.

In the above experiments, the performance of the disposable index confirms the theoretical analyses in Section 8.3. The disposable index outperformed the B^x-tree on location updates and continuous query processing, with similar search performance. The scalability of the disposable index has been validated for mobility, phase, Q_size, and Q_ratio. The performance of update operations is mainly determined by the mobility, while the cost of a window search depends on the Q_size and phase.

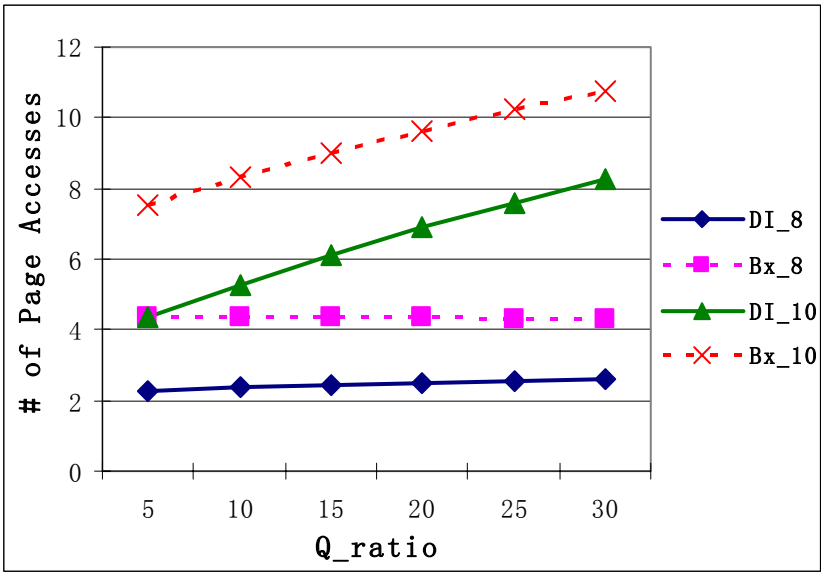


Figure 79. Movement I/O vs. Q_ratio.

8.5 Conclusion

This chapter proposes a framework for moving object management with optimized update cost and independent indexing components. The proposed disposable index eliminates the deletion of obsolete locations on indexing trees and enables parallel search. Snapshot query operations and continuous query operations are supported in the disposable index. Both the theoretical analysis and experimental evaluation have been conducted to show 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 the ability of parallel processing. Future efforts could be devoted to further evaluating the performance of this framework on parallel computing hardware. Expanding this framework to other query types, including nearest neighbor query, and aggregation queries would also be an interesting direction.

Chapter 9. COMPLETED WORK AND FUTURE DIRECTIONS

This Ph.D. research focuses on designing concurrent spatial operations for spatial access methods. Following the two categories of spatial indexing methods, efficient concurrent access frameworks are proposed for applications in multi-user environments. In addition to the development of concurrent access frameworks for stationary objects, concurrent operations on moving objects are designed in order to expand these frameworks to the emergent applications in GIS and mobile networks. Empirical results have shown the efficiency and scalability of the proposed frameworks.

9.1 Research Achievements

This research has followed two major branches, concurrent operations on R-tree-based spatial index, and concurrent operations on linear spatial indexing approaches, as illustrated in Figure 4. Specifically, there are several research topics on each branch that are studied in this research. These topics as well as the corresponding tasks are listed as follows. The outputs of these topics are listed in Table 13.

A. Concurrent Access Framework Based on R-trees

ZR+-tree indexing structure

A Zero-overlap R+-tree (ZR+-tree) based on object clipping technique has been proposed to remove some limitations on R+-tree, together with the fundamental operations on it. Spatial operations on the ZR+-tree utilizes sophisticated insertion and splitting mechanisms to optimize the indexing tree and avoid the insertion deadlock that may happen on the R+-tree. With the refined tree structure and the risk-free tree construction, the ZR+-tree outperforms the R-tree and R+-tree in search-dominated environments. The experiments on both synthetic data and real datasets verify the outstanding performance of the ZR+-tree in terms of the I/O cost and standard deviation for query processing.

Concurrency Control Protocol GLIP

A concurrency control protocol, namely Granular Locking on clIPping indexing (GLIP), has been designed to provide serializable isolation, data consistency, and additional deadlock free on indexing trees with object clipping. This protocol is implemented based on the ZR+-tree, and efficiently supports the fundamental spatial operations, including search, insertion, and deletion. Theoretical analysis, as well as sets of experiments on both synthetic data and real data, show that the concurrent framework based on the ZR+-tree and GLIP outperforms the existing concurrency protocol on the R-tree, in terms of the throughput

of concurrent spatial operations.

Concurrency Control Protocol for Continuous Queries C^3 based on R-trees

Continuous query on moving objects is a challenge to concurrency control protocols, because they need to keep monitoring a certain spatial area / subset of data. On the other hand, frequent updates on R-trees require efficient buffering techniques, which make the index structure too complex for existing concurrency control protocols. A solution for concurrent continuous operations is proposed for the spatial data access platform of R-tree with lazy-update. Modifications generalized from existing R-tree buffering approaches are applied to the R-tree to facilitate the frequent update operations. Meanwhile, concurrent continuous query processing has been designed to protect the results from these update operations, allowing as much concurrent update operations to be processed as possible. Both analysis and experiments are presented to verify this approach.

B. Concurrent Access Framework Based on Linear Spatial Index

Efficient kNN search on SFC and B+-tree with generic optimizations

For the spatial indexing based on SFC and B-trees, an efficient incremental nearest neighbor (INN) search which utilizes the grid structure in SFC space is designed. Three general optimizations, multiple SFCs, query composition, and bitmap, are proposed to improve the query performance by considering the synergy between SFC and B-trees. The experiments show that the INN search is scalable and takes advantage from these optimization techniques.

Concurrent Location Management (CLAM) on SFC and B^{link}-tree

Spatial location updates on SFC and the B^{link}-tree cannot be efficiently protected in multiple-user environments by existing concurrency control protocols on the B^{link}-tree. Concurrent spatial operations, including location update and range query, are designed to provide serializable isolation and data consistency in this type of spatial indexing structures. These concurrent operations merge the link-based and lock-coupling protocols to form up a concurrent access framework CLAM on SFC and the B^{link}-tree. Theoretical analysis and experiments have been conducted to assess the performance of CLAM in terms of the spatial operation throughput.

Concurrent Continuous Query on Moving Objects based on SFC and B^{link}-tree

The spatial access framework with SFC and the B^{link}-tree is suitable for point data, and is efficient for updating. Continuous query on moving objects can benefit from the efficient updating of this spatial indexing method. However, designing a concurrency control protocol for continuous query in this

framework is a challenge, because a certain spatial area / subset of data should be continuously protected, which may degrade the throughput of the spatial operations. Solution for concurrent continuous range query is proposed by extending the spatial data access platform of CLAM. Experiments on real datasets have been conducted to verify this approach.

Improved Concurrent Continuous Query Processing on Moving Objects with Known Velocity

Moving object index with velocity information has been shown to significantly reduce the location update cost. Such index methods include B^x-tree and BB^x-tree. By expanding the concurrent continuous query processing to these index methods, higher throughput than with a pure B^{link}-tree has been demonstrated.

C. Generic Spatio-temporal Access Framework

Disposable Index

Deleting obsolete locations from spatial indices consumes significant I/O cost in moving object management. A generic framework for spatial access framework, Disposable Index, has been proposed to eliminate the deletion of obsolete locations. Furthermore, it has a parallel indexing structure that supports parallel search on the moving object index. This framework can work with either a linear spatial index or spatial index trees, and can reduce the overhead of concurrency control.

Table 13. Research Topics and Outputs.

Topics		Outputs
Branch 1 ***(A)***	**Concurrent Access Framework Based on R-trees**	
ZR+-tree indexing structure		***TKDE 09'***
Concurrency protocol GLIP		***TKDE 09'***
Concurrent continuous query processing C^3		***Submitted to ICDE 10'***
Branch 2 ***(B)***	**Concurrent Access Framework Based on B-trees**	
Efficient kNN search on spatial indexing methods with SFC		***Submitted to J. Geoinformatica***
Concurrent location management CLAM		***ACM GIS 07'***
Concurrent continuous query based on linear spatial index		***MDM 09'***
Concurrent continuous query based on known velocity		***To be submitted to TKDE***
Generic ***(C)***	Disposable Index	***To be submitted to EDBT10'***

The major contributions of this research are as follows:

- Propose efficient indexing and query processing approaches for popular spatial access frameworks

- Design concurrent protocols for stationary and moving objects/queries
- Assure serializable isolation, data consistency, and deadlock-freedom
- Validate the proposed approaches with correctness proofs and benchmark data sets
- Significantly enhance the applicability of spatial data management in real-world scenarios

9.2 Future Directions

Further efforts could focus on extending the developed concurrent access frameworks to perform complex operations, such as range aggregation, nearest neighbor queries, and trajectory queries. These operations usually need to access a larger set of data than location updates and window queries. Therefore, optimizations are needed for the existing access frameworks to efficiently handle these operations. More real-world datasets, e.g., the Reality Mining Dataset [73], could be used to further evaluate the performance and applicability of these frameworks.

Interesting topics that could be extended from this research include concurrent operations on distributed spatial databases, because distributed databases raise challenges to the existing concurrency control protocols, and require different performance measures. Another future direction could be extending the concurrent spatial operations to graph networks. Graph networks calculate distance using the shortest route in the network rather than the Euclidian distance. Therefore optimizations are required to ensure the efficiency and correctness of concurrent operations.

9.3 Publications

Journal Publications

Jing Dai, Chang-Tien Lu, "Concurrent Continuous Query Processing on Linear Spatial Indices," to be submitted to *IEEE Transaction on Knowledge and Data Engineering.*

Jing Dai, Chang-Tien Lu, Ying Jin, "Similarity Search on Linear Spatial Indices," Submitted to *Journal of Geoinformatica*.

Chang-Tien Lu, Jing Dai, Ying Jin, Janak Mathuria, "GLIP: A Concurrency Control Protocol for Clipping Indexing," *IEEE Transaction on Knowledge and Data Engineering, vol. 21, No. 5, pp. 712-728, May, 2009*.

Conference Publications

Jing Dai, Chang-Tien Lu, "Disposable Index for Managing Moving Objects," to be submitted to *International Conference on Extending Database Technology, Lausanne, Switzerland, March, 2010.*

Jing Dai, Chang-Tien Lu: "C3: Concurrency Control on Continuous Queries over Moving Objects," Submitted to *IEEE International Conference on Data Engineering, Long Beach, CA, March, 2010*.

Jing Dai, Chang-Tien Lu, Lien-Fu Lai: "A Concurrency Control Protocol for Continuously Monitoring Moving Objects," *Proceedings of the 10th International Conference on Mobile Data Management, pp. 132-141, Taipei, Taiwan, May, 2009.*

Chang-Tien Lu, Arnold P. Boedihardjo, Jing Dai, Feng Chen, "HOMES: Highway Operation Monitoring and Evaluation System," *Proceedings of the 16th ACM International Conference on Advances in Geographic Information Systems, pp. 529-530, Irvine, CA, November, 2008.*

Jing Dai, Chang-Tien Lu, "CLAM: Concurrent Location Management for Moving Objects," *Proceedings of the 15th ACM International Symposium on Advances in Geographic Information Systems, pp. 38-47, Seattle, WA, November, 2007*.

Ying Jin, Jing Dai, Chang-Tien Lu, "Spatial-Temporal Data Mining in Traffic Incident Detection," *Proceedings of SIAM Data Mining Conference, Spatial Data Mining Workshop, Bethesda, MD, April, 2006*.

Janak Mathuria, Chang-Tien Lu, Jing Dai, "An Efficient Data Model for Sensor Networks," *the 1st International Workshop on Data Mining in conjunction with 8th Joint Conference on Information Sciences, Salt Lake City, UT, July, 2005.*

Demos

Jing Dai, Arnold Boedihardjo: "Highway Performance Monitoring and Evaluation System," *Virginia Graduate Research Symposium, Richmond, VA, February, 2009*.

Jing Dai, Arnold Boedihardjo, Chang-Tien Lu, Feng Chen: "HOMES: Highway Operation Monitoring and Evaluation System," *Summit on the National Academy of Engineering's Grand Challenges, 2nd prize in poster contest, Durham, NC, March, 2009*.

Jing Dai: "Large Scale Spatial Data Management and Its Application in ITS," *Paul E. Torgersen Research Award Nominated Poster, Blacksburg, VA, April, 2009*.

Book Chapters

Jing Dai, Chang-Tien Lu, "Concurrency Control for Spatial Access Methods," *Encyclopedia of Geographical Information Science,* Shashi Shekhar and Hui Xiong (eds.), pp. 124-128, Springer, ISBN 978-0-387-30858-6, 2008.

BIBLIOGRAPHY

[1] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Boston, MA, USA, pp. 47-57, June 18-21, 1984.

[2] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transaction of Database System,* vol. 9, no. 1, pp. 38-71, 1984.

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

[4] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and Update Efficient B+-Tree Based Indexing of Moving Objects," *in Proceedings of International Conference on Very Large Data Bases*, Toronto, Canada, pp. 768-779, August 29-September 3, 2004.

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

[6] N. Roussopoulos, S. Kelly, and F. Vincent, "Nearest Neighbor Queries," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, pp. 71-79, May 22-25, 1995.

[7] G. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Transactions of Database Systems,* vol. 24, no. 2, pp. 265-318, 1999.

[8] S. I. Song, Y. H. Kim, and J. S. Yoo, "An Enhanced Concurrency Control Scheme for Multidimensional Index Structure," *IEEE Transactions on Knowledge and Data Engineering,* vol. 16, no. 1, pp. 97-111, 2004.

[9] K. Chakrabarti and S. Mehrotra, "Dynamic Granular Locking Approach to Phantom Protection in R-trees," *in Proceedings of IEEE International Conference on Data Engineering*, Orlando, FL, USA, pp. 446-454, February 23-27, 1998.

[10] K. Chakrabarti and S. Mehrotra, "Efficient Concurrency Control in Multi-dimensional Access Methods," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, USA, pp. 25-36, June 1-3, 1999.

[11] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A Dynamic Index for Multi-dimensional Objects," *in Proceedings of International Conference on Very Large Data Bases*, Brighton, England, pp. 507-5181987.

[12] V. Gaede and O. Gunther, "Multidimensional Access Methods," *ACM Computing Surveys,* vol. 30, no. 2, pp. 170-231, 1998.

[13] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Paris, France, pp. 321-330, June 15-17, 2004.

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

[15] H. Hu, J. Xu, and D. L. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries

over Moving Objects," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Baltimore, MD, USA, pp. 479-490, June 14-16, 2005.

[16] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo, "Supporting Frequent Updates in R-Trees: A Bottom-Up Approach," *in Proceedings of International Conference on Very Large Data Bases*, Berlin, Germany, pp. 608-619, September 9-12, 2003.

[17] X. Xiong and W. G. Aref, "R-trees with Update Memos," *in Proceedings of IEEE International Conference on Data Engineering*, Atlanta, GA, USA, pp. 22-31, April 3-8, 2006.

[18] L. Biveinis, S. Saltenis, and C. S. Jensen, "Main-memory Operation Buffering for Efficient R-tree Update," *in Proceedings of International Conference on Very Large Data Bases*, Vienna, Austria, pp. 591-602, September 23-27, 2007.

[19] B. Lin and J. Su, "Handling Frequent Updates of Moving Objects," *in Proceedings of ACM International Conference on Information and Knowledge Management*, Bremen, Germany, pp. 493-500, October 31-November 5, 2005.

[20] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve," *IEEE Transactions on Knowledge and Data Engineering,* vol. 13, no. 1, pp. 124-141, January/February 2001.

[21] M. F. Mokbel, W. G. Aref, and I. Kamel, "Analysis of Multi-dimensional Space-Filling Curves," *GeoInformatica, An International Journal on Advances of Computer Science for Geographic Information System,* vol. 7, no. 3, pp. 179-209, September 2003.

[22] S. Liao, M. A. Lopez, and S. T. Leutenegger, "High Dimensional Similarity Search With Space Filling Curves," *in Proceedings of IEEE International Conference on Data Engineering*, Heidelberg, Germany, pp. 615-622, April 2-6, 2001.

[23] H. Chen and Y. Chang, "Neighbor-finding based on Space-filling Curves," *Information System,* vol. 30, pp. 205-226, 2005.

[24] J. A. Orenstein, "Spatial Query Processing in an Object-Oriented Database System," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Washington D.C., USA, pp. 326-336, May 28-30, 1986.

[25] J. K. Lawder and P. J. H. King, "Querying Multi-dimensional Data Indexed Using the Hilbert Space-Filling Curve," *SIGMOD Record,* vol. 30, no. 1, pp. 19-24, 2001.

[26] C. Faloutsos, "Gray Codes for Partial Match and Range Queries," *IEEE Transactions on Software Engineering,* vol. 14, no. 10, pp. 1381-1393, 1988.

[27] W. d. Jonge and A. Schiff, "Concurrent Access to B-trees," *in Proceedings of PARBASE International Conference on Databases, Parallel Architectures and Their Applications*, Miami Beach, FL, USA, pp. 312-320, March 7-9, 1990.

[28] V. Lanin and D. Shasha, "A Symmetric Concurrent B-tree Algorithm," *in Proceedings of Fall Joint Computer Conference*, Dallas, TX, USA, pp. 380-389, November 2-6, 1986.

[29] P. Lehman and S. Yao, "Efficient Locking for Concurrent Operations on B-trees," *ACM Transactions on Database Systems,* vol. 6, no. 4, pp. 650-670, 1981.

[30] 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 International Conference on Mobile Data Management*, Ayia Napa, Cyprus, pp. 59-66, May 9-13, 2005.

[31] 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, pp. 321-330, Jun. 13-18, 2004.

[32] D. Lomet, "Key Range Locking Strategies for Improved Concurrency," *in Proceedings of International Conference on Very Large Data Bases*, Dublin, Ireland, pp. 655-6641993.

[33] C. Mohan, "ARIES/KVL: A Key Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes," *in Proceedings of International Conference on Very Large Data Bases*, Brisbane, Australia, pp. 392-4051990.

[34] D. Greene, "An Implementation and Performance Analysis of Spatial Data Access Methods," *in Proceedings of IEEE International Conference on Data Engineering*, Los Angeles, CA, USA, pp. 606-615, February 6-10, 1989.

[35] D. Zhang and T. Xia, "A Novel Improvement to the R*-tree Spatial Index Using Gain/Loss Metrics," *in Proceedings of ACM International Symposium on Advances in Geographic Information Systems*, Washington DC, USA, pp. 204-2132004.

[36] M. Abdelguerfi, J. Givaudan, K. Shaw, and R. Ladner, "The 2-3TR-tree, a Trajectory-Oriented Index Structure for Fully Evolving Valid-Time Spatio-Temporal Datasets," *in Proceedings of ACM GIS International Symposium on Advances in Geographic Information Systems*, McLean, VA, USA, pp. 29-34, November 8-9, 2002.

[37] L. Shou, Z. Huang, and K.-L. Tan, "The Hierarchical Degree-of-Visibility Tree," *IEEE Transactions on Knowledge and Data Engineering,* vol. 16, no. 11, pp. 1357-1369, November 2004.

[38] Y. Tao and D. Papadias, "Performance Analysis of R*-Trees with Arbitrary Node Extents," *IEEE Transactions on Knowledge and Data Engineering,* vol. 16, no. 6, pp. 653-668, June 2004.

[39] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee, "Performance Evaluation of Main-Memory R-tree Variants," *in Proceedings of International Symposium on Spatial and Temporal Databases*, Santorini Island, Greece, pp. 10-27, July 24-27, 2003.

[40] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, USA, pp. 322-331, May 23-25, 1990.

[41] E. G. Hoel and H. Samet, "A Qualitative Comparison Study of Data Structures for Large Line Segment Databases," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, San Diego, CA, USA, pp. 205-214, June 2-5, 1992.

[42] L. Biveinis, S. Saltenis, and C. S. Jensen, "Main-memory Operation Buffering for Efficient R-tree Update " *in Proceedings of International Conference on Very Large Data Bases*, Vienna, Austria, pp. 591-6022006.

[43] J. K. Chen, Y. F. Huang, and Y. H. Chin, "A Study of Concurrent Operations on R-Trees," *Information Science,* vol. 98, no. 1-4, pp. 263-300, 1997.

[44] V. Ng and T. Kamada, "Concurrent Accesses to R-Trees," *in Proceedings of Symposium on Large Spatial Databases*, Singapore, pp. 142-1611993.

[45] K. V. R. Kanth, D. Serena, and A. K. Singh, "Improved Concurrency Control Techniques for Multi-Dimensional Index Structures," *in Proceedings of Symp. Parallel and Distributed Processing*,

pp. 580-5861998.

[46] M. Kornacker and D. Banks, "High-Concurrency Locking in R-Trees," *in Proceedings of International Conference on Very Large Data Bases*, Zurich, Switzerland, pp. 134-145, September 11-15, 1995.

[47] M. Kornacker, C. Mohan, and J. Hellerstein, "Concurrency and Recovery in Generalized Search Trees," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, USA, pp. 62-72, May 13-15, 1997.

[48] C. Mohan and F. Levin, "ARIES/IM: an Efficient and High Concurrency Index Management Method Using Write-ahead Logging," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, San Diego, California, United States, pp. 371-3801992.

[49] A. Biliris, "Operation Specific Locking in B-trees," *in Proceedings of International Conterence on Principles of Database Systems*, San Diego, California, United States, pp. 159-1691987.

[50] G. Peano, "Sur Une Courbe Qui Remplit Toute Une Air Plaine," *Math. Ann.,* vol. 36, pp. 157-160, 1890.

[51] C. Faloutsos, "Multiattribute Hashing Using Gray Codes," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Washington, D.C., USA, pp. 227-238, May 28-30, 1986.

[52] D. Hilbert, "Ueber Stetige Abbildung Einer Linie Auf Ein Flashenstuck," *Mathematishe Annalen,* pp. 459-460, 1981.

[53] H. V. Jagadish, "Linear Clustering of Objects with Multiple Attributes," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, USA, pp. 332-342, May 23-25, 1990.

[54] D. Lee and H. Kim, "An Efficient Technique for Nearest-Neighbor Query Processing on the SPY-TEC," *IEEE Transactions on Knowledge and Data Engineering,* vol. 15, no. 6, pp. 1472-1488, November/December 2003.

[55] T. Seidl and H.-P. Kriegel, "Optimal Multi-Step k-Nearest Neighbor Search," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 154-1651998.

[56] I. Jaluta, S. Sippu, and E. Soisalon-Soininen, "Concurrency Control and Recovery for Balanced B-link Trees," *VLDB Journal,* vol. 14, no. 2, pp. 257-277, 2005.

[57] V. W. Setzer and A. Zisman, "New Concurrency Control Algorithms for Accessing and Compacting B-trees," *in Proceedings of International Conference on Very Large Data Bases*, Santiago de Chile, Chile, pp. 238-248, September 12-15, 1994.

[58] V. Srinivasan and M. Carey, "Performance of B±tree Concurrency Control Algorithm," *VLDB Journal,* vol. 2, no. 4, pp. 416-425, 1993.

[59] J. K. Chen and Y. H. Chin, "A Concurrency Control Algorithm for Nearest Neighbor Query," *Information Science,* vol. 114, pp. 187-204, 1999.

[60] C. S. Jensen, D. Tielsytye, and N. Tradilauskas, "Robust B+-Tree-Based Indexing of Moving Objects," *in Proceedings of International Conference on Mobile Data Management*, Nara, Japan, pp. 12-20, May 9-13, 2006.

[61] M. L. Yiu, Y. Tao, and N. Mamoulis, "The B^{dual}-Tree: Indexing Moving Objects by Space Filling Curves in the Dual Space," *VLDB Journal,* vol. 17, no. 3, pp. 379-400, May 2008.

[62] 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, pp. 479-490, June 14-15, 2005.

[63] J. T. Robinson, "The K-D-B-tree: a search structure for large multidimensional dynamic indexes," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 10-181981.

[64] P. S. Yu, K.-L. Wu, K.-J. Lin, and S. H. Son, "On Real-time Databases: Concurrency Control and Scheduling," *in Proceedings of IEEE*, pp. 140-1571994.

[65] S. Guha, R. Rastogi, and K. Shim, "CURE: An Efficient Clustering Algorithm for Large Databases," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, Seattle, WA, USA pp. 73-84, June 2-4, 1998.

[66] F. Li and G. Kollios, "Real Datasets for Spatial Databases: Road Networks and Points of Interest," *http://cs-people.bu.edu/lifeifei/SpatialDataset.htm*, Accessed in 2006.

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

[68] B. Zheng, W.-C. Lee, and D. L. Lee, "Spatial Queries in Wireless Broadcast Systems," *Wirel. Netw.,* vol. 10, pp. 723-736, 2004.

[69] S. Berchtold, C. Bohm, D. A. Keim, and H. P. Kriegel, "A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space," *in Proceedings of ACM Symposium on Principles of Database Systems*, Tucson, AZ, USA, pp. 78-86, May 12-14, 1997.

[70] A. Henrich, "A Distance-scan Algorithm for Spatial Access Structures," *in Proceedings of ACM Workshop on Advances in Geographic Information Systems*, Gaithersburg, MA, USA, pp. 136-143, December 1-2, 1994.

[71] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing Bitmap Indices with Efficient Compression," *ACM Transactions on Database Systems,* vol. 31, no. 1, pp. 1-38, March 2006.

[72] 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, pp. 292-299, November 7-9, 2007.

[73] MIT, "Reality Mining Dataset," *http://reality.media.mit.edu/dataset.php*, Accessed in 2009.