S^{e}TPR*-tree: Efficient Buffering for Spatiotemporal Indexes Via Shared Execution

THI NGUYEN, ZHEN HE, YI-PING PHOEBE CHEN

Department of Computer Science and Computer Engineering, La Trobe University, VIC 3086, Australia
Email: nt2nguyen@students.latrobe.edu.au, z.he@latrobe.edu.au, Phoebe.Chen@latrobe.edu.au

In this paper, we study the problem of efficient spatiotemporal indexing of moving objects. In order to reduce the frequency of object location updates, a linear motion model is used to model the near future location of moving objects. A number of existing spatiotemporal indexes have already been proposed for indexing these models. However, these indexes are either designed to offer high query performance or high update performance. Therefore, they are all ill suited to handle situations where both queries and updates arrive at a high rate. In this paper, we propose the S^{e}TPR*-tree which extends the TPR*-tree to more efficiently use a limited sized RAM buffer for processing queries in batches and rapidly arriving updates. We provide both theoretical and empirical evidence of the effectiveness of the S^{e}TPR*-tree in improving query and update performance. We have conducted extensive experiments using a recognized spatiotemporal benchmark on a solid state drive. The S^{e}TPR*-tree simultaneously outperforms the best tested index optimized for queries by up to a factor of 5.6 for query I/O and outperforms the best tested index for updates by up to a factor of 11.5 for update I/O.

Keywords: Spatiotemporal databases, Indexing, Query processing

1. INTRODUCTION

An increasing number of mobile devices such as mobile phones and car navigation systems are becoming GPS-enabled and capable of reporting their current location wirelessly to a central server. This allows the design of many different applications which involve querying the current or near future locations of the mobile devices.

A popular method to reduce the report rate from mobile devices is to use a linear motion-based model to model the near future location of moving objects. The model consists of the initial extent of the object and a velocity vector. An update is issued by the object when its velocity changes. A number of index structures have been proposed to index and query the trajectory information. These indexes include the TPR-tree [1], TPR*-tree [2], B^{dual}-tree [3], B^x-tree [4], ST^{2}B-tree [5], etc. These index structures are optimized for one-at-a-time querying and updating. However, there are many situations where a potentially large number of queries and updates need to be executed on the index at the same time. We call these "batch" queries and updates.

Queries and updates need to be queued and executed in batch when they arrive faster than the processing rate. For example, a large stream of index update requests can arrive in a very short time period as a result of many objects invalidating their linear motion models around the same time. In our experiments, we found that the fastest existing spatiotemporal index for updates takes around 6 ms to process an individual update when indexing 1 million objects. This means that if more than 166 out of the 1 million objects need to be updated within 1 second, then updates and later arriving queries need to be queued and processed later in a batch. We envisage this situation to occur often in a real system using these indexes. We therefore propose algorithms for the fast batch processing of queries and updates for spatiotemporal indexes.

Another situation where batch query execution is particularly useful is for processing continuous intersection joins [6], where one set of objects needs to be regularly joined with another by effectively issuing a batch of range queries simultaneously on a spatiotemporal index.

It is important to note that there is a tradeoff between response time and throughput when queries are executed in batches. Executing batches of queries can improve the efficiency of query execution, leading to increased throughput. However, this will also lead to increased response time. In this paper, we mainly focus on improving throughput since we are interested in situations where queries arrive too fast for the system to handle. If queries are not processed fast enough, the system will never catch up to the speed of query arrival, hence in this situation, it is more desirable to maximize throughput than minimize response time. We have measured this tradeoff between response time and throughput in our experiments.

All existing spatiotemporal indexes struggle to cope with the situation when many queries and updates occur in a short period of time for the following reasons:

- ineffective RAM buffer usage resulting in high I/O costs for each query and update executed.
- optimizing for either query or update performance but not both.

Due to the slow speed of secondary storage (either hard disk drives or solid state drives) compared to RAM, secondary storage indexes use a RAM buffer to cache recently used index nodes and thereby hide some of the high access costs of secondary storage. The buffer only works well when there is high temporal locality. However, in existing systems, the degree of temporal locality is essentially determined by luck, namely the order in which the queries arrive. High temporal locality occurs when spatially close queries happen to arrive in close succession.

We propose the S^{e}TPR*-tree which uses three techniques to reorder index node references for batched queries (both range queries and k nearest neighbour queries) and thereby increase temporal locality. The three techniques are *shared query execution*, *shared deletion execution* and *proximity ordered insertion*. Shared query execution groups queries together and executes them in a batch so that the minimum number of page loads are performed to execute the entire batch of queries. Shared deletion works by first caching all deletions in the RAM buffer and then executing the deletions in batch when the RAM is full. Finally, we also cache object insertions in RAM and then insert them into the tree in proximity order when the RAM buffer is full.

The idea of improving temporal locality by performing query execution, deletion and insertion in batch has been proposed in the area of indexing and querying static objects using R-trees [7, 8, 9]. However, none of these ideas have been studied comprehensively in the context of *spatiotemporal* indexes. This paper first analyzes the negative consequences of naively applying these batch execution techniques into R-tree based spatiotemporal indexes. Based on this analysis, we propose the S^{e}TPR*-tree which incorporates tailored batch execution algorithms for improving the query and update performance of the TPR*-tree (a state-of-the-art spatiotemporal index). The *tailored* algorithms are designed in an *integrated* way to maximize the utilization of the limited RAM buffer by using *dynamic buffer allocation*. The S^{e}TPR*-tree algorithms can be applied to any other R-tree based spatiotemporal index such as the TPR-tree.

In addition to the above, this paper offers the first detailed performance study of the effect of optimizing temporal locality in the context of spatiotemporal indexes. The main results of the performance study are as follows:

- The high temporal locality update algorithms of the S^{e}TPR*-tree outperforms the closest competitor for updates, RUM*-tree (the RUM-tree [8] update algorithms applied to the TPR*-tree) by a factor 11.5 for update I/O.
- Although the batched update algorithms of the S^{e}TPR*-tree cause queries to slow down due to delaying updates (frequent updates are essential for re-optimizing the tree for querying), the shared query execution more than compensates for this slow down. The result is that the shared range query execution algorithm of the S^{e}TPR*-tree outperforms its closest competitor, the TPR*-tree (best for queries) by up to a factor of 5.6 for query I/O.
- The S^{e}TPR*-tree only needs to process a batch of 5 queries or more to outperform all tested algorithms for query I/O performance.
- Shared deletion execution and proximity ordered insertion, when used together, result in the largest reduction in update I/O. However on their own, shared deletion execution is more effective than proximity ordered insertion at reducing update I/O.
- Proximity ordered insertion significantly outperforms arrival ordered insertion.
- The high temporal locality operations of the S^{e}TPR*-tree make it much less sensitive to changes in the characteristics of the data and query sets. The query and update I/O performance per operation of the S^{e}TPR*-tree stays near constant, with varying update frequency, query size and maximum velocity of moving objects. This contrasts with other tested indexes whose performance deteriorated significantly with decreased update frequency, enlarged query size and increased maximum velocity of moving objects.

This paper makes five key contributions. Specifically:

- we propose a spatiotemporal index called S^{e}TPR*-tree which is simultaneously optimized for executing batches of queries and individually arriving updates;
- the S^{e}TPR*-tree reorders the query and update reference stream to the spatiotemporal index and thereby improves the temporal locality of the reference stream;
- we theoretically prove that our S^{e}TPR*-tree achieves minimum buffer miss rates;
- we integrate our proposed techniques for increasing temporal locality with dynamic RAM buffer allocation to make the best use of the limited sized RAM buffer; and
- we conduct an extensive experimental study to compare our S^{e}TPR*-tree with three rival state-of-the-art spatiotemporal indexes.

The remainder of the paper is organized as follows. Section 2 describes the related work which includes a detailed look at R-tree based spatiotemporal indexes; Section 3 describes the benefits of shared execution for reducing I/O costs; Section 4 describes our proposed S^{e}TPR*-tree; Section 5 describes the experimental setup used to test our S^{e}TPR*-tree; Section 6 shows the experimental results and analysis for comparing the effectiveness of our S^{e}TPR*-tree with existing indexes; and finally, in Section 7, we conclude the paper and outline directions for future work.

2. RELATED WORK

In this section, we start by describing the existing work on R-tree based spatiotemporal indexes and its dependence on updates for reducing MBR overlap. Next, we present existing work on supporting efficient updates on R-tree based indexes. We then present existing work on bulk loading R-tree and TPR-tree indexes. Finally, we briefly describe existing work on B$^+$-tree based spatiotemporal indexes.

2.1. R-tree based spatiotemporal indexes

An established approach to index spatiotemporal data is to use the R*-tree to index the extents of objects and their current velocity. These indexes include the TPR-tree [1] and the improved TPR*-tree [2]. They work by grouping object extents at the reference time into minimum bounding rectangles (MBRs). Figure 1(a) shows the objects O_1, O_2 and O_3 grouped into the same MBR in node N_1. Accompanying the MBRs are the velocity bounding rectangles (VBRs) which represent the expansion of the MBRs with time according to the velocity vectors of the constituent objects. The rate of expansion in each direction is equal to the maximum velocity among the constituent objects in the corresponding direction. A negative velocity value implies that the velocity is towards the negative direction of the axis. For example, in Figure 1(a), we can see the hollow arrow on the left of node N_1 has a value of -2. This is because the maximum velocity value of the constituent objects in the left direction is 2.

The MBR and VBR structure described can be extended by replacing the constituent object extents with smaller MBRs. This, when recursively applied, creates a hierarchical tree structure. The tree structure is identical to the classic R-tree [10], the only difference being the algorithms used to insert, delete and query the tree also need to take the velocity information into consideration. The TPR-tree and TPR*-tree modify the R*-tree insertion/deletion and query algorithms to operate on the tree.

2.1.1. Issue of overlapping MBRs

An important phenomenon that occurs as a result of using the R-tree structure to index this type of spatiotemporal information is that the MBRs can rapidly expand with time and thus cause an increased overlap among MBRs. For example, Figure 1(b) shows the expansion of the MBRs of nodes N_1 and N_2 at time 1 from the MBRs of the corresponding nodes at time 0 (Figure 1(a)). To reduce MBR expansion, the TPR-tree and TPR*-tree tighten the MBRs of tree nodes following object location updates. For example, Figure 1(c) shows the deletion of O_1 followed by the tightening of the MBR for N_1.

The above example shows the importance of tightening the MBR to reduce the amount of overlap for the TPR-tree and TPR*-tree. Overlap increases query execution cost. For example in Figure 1(b), query Q_1 needs to inspect the objects in both N_1 and N_2. In contrast, in Figure 1(c) the same query Q_1 only needs to inspect the objects in N_2, since the MBR of N_1 has been tightened.

It is important to note that MBR tightening only occurs as a result of updates (either deletions or insertions) in the TPR-tree and TPR*-tree. Therefore, frequent update operations on the index are vital to its query performance. However, the downside of frequent updates is the high I/O costs. This is due to the need to update multiple MBRs and possible tree restructuring. A naive solution for reducing update I/O is to use one of the existing methods [7, 8, 9] that buffer deletion and/or insertion in RAM before applying them in batch to the disk-based R-tree. Another way of delaying the application of updates to disk, is to attempt to restrict updates to leaf nodes only by using a bottom-up update approach [11, 12]. However, following the analysis above, delaying the application of updates to disk will increase query I/O costs.

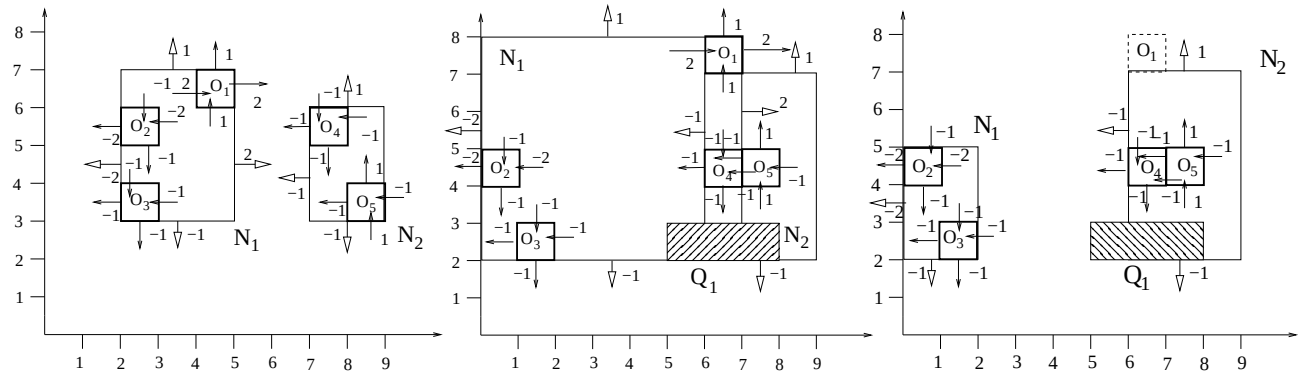
The above raises an interesting dilemma, namely, delaying updates reduces update I/O but at the same time, also increases read I/O costs of queries. Our proposed solution which uses shared execution to execute queries in batch overcomes the negative effects of delayed updates for the following reasons: increased temporal locality; and reduced sensitivity to MBR overlap. In Section 3, we will show how shared execution can increase temporal locality and how it can reduce sensitivity to MBR overlap. Both of these factors contribute to the overall reduction in I/O costs during querying.

2.2. Efficient updates on R-tree based indexes

In this section, we describe in more detail how existing work can reduce the cost of updates on R-tree based indexes and why they result in poor query performance when used for indexing spatiotemporal data.

One way to reduce update costs is to perform deletions followed by bottom-up insertions [11, 12]. In memory, indexes are used to directly identify leaf nodes containing the object to be deleted. Next, an attempt is made to directly reinsert the object in the same leaf node involved in the deletion without doing a top-down traversal. One of the key benefits of this approach for indexing stationary objects is that non-leaf nodes can be left untouched. However, when this is applied to spatiotemporal indexes, this at best, delays MBR tightening and at worst, never tightens any MBRs. The reason is that tightening of MBRs can only be done by updating non-leaf nodes, since the MBR of all nodes are stored in non-leaf nodes.

The RUM-tree [8] (which stands for R-tree with update memo), achieves high update performance by buffering deletions in main memory before applying them to disk in batch. The R^R-tree [7] proposed by Biveinis et al. buffers both insertions and deletions. When the operation buffer is full, some or all of its operations are applied to disk in bulk. As mentioned earlier, these approaches attempt to delay updates to the disk tree which results in delayed tightening of the MBRs and hence, increased overlapping of MBRs. As



(a) MBRs and VBRs at time 0 (b) MBRs and VBRs at time 1 (c) Tightened MBR after deleting O_1 at time 1

FIGURE 1. Example of MBRs of a TPR-tree growing with time and subsequent tightening of node N_1's MBR after the deletion of object O_1

a consequence, when applied to spatiotemporal indexes, it results in reduced query performance. Our work differs from these in two respects. First, our shared query execution is less sensitive to the negative effects of overlapping MBRs, as explained in Section 3.2. Second, unlike Biveinis et al. [7], we perform proximity ordered insertion instead of batched insertion which results in a more optimized tree (see Section 4.4 for a more detailed explanation).

2.3. Bulk loading R-trees and TPR-trees

In this section, we present techniques designed for the bulk loading of R-trees and TPR-trees. Most of the existing work in bulk loading R-trees and TPR-trees is designed to speed up the bulk insertion of a group of *s*tatic data objects [13, 14, 15, 16, 17, 18, 19, 20] into an empty tree. Therefore, they are not designed to handle continuously arriving updates and hence, are not useful for our purposes.

Arge et al. [21] propose the use of a simple lazy buffering technique for performing bulk operations on the R-trees. In contrast to the above work, they support performing batched insertions, deletions and queries on an already existing tree. They attach buffers to all R-tree nodes at certain levels of the tree. These buffers are then used to cache tree operations and apply them lazily to nodes lower down the tree. However, their buffers reside on *disk*. Therefore, using their approach introduces extra read and write IO when accessing the disk buffer. In contrast, we buffer batch operations in RAM and therefore do not incur any extra IO overheads.

Zhou et al. [22] apply a similar technique to Arge et al. [21] but on main memory indexes instead of disk resident indexes. Therefore, their buffers reside in main memory instead of disk. They support both batched insertions and updates. The key difference between their work and ours is that we focus on disk-resident spatiotemporal indexes instead of memory resident indexes.

Lin et al. [9] combine disk buffered lazy bulk insertions with RAM buffered bottom-up batched deletions. This technique introduces additional read and write I/O for the disk buffers for insertions. Its query performance is low due to its approach of delaying updates.

2.4. B$^+$-tree based spatiotemporal indexes

The R-tree based spatiotemporal indexes like TPR*-tree have high query performance if updates are performed frequently. However, they suffer from low update performance due to expensive node splitting and merging operations. Storing the spatiotemporal data inside a B$^+$-tree overcomes this problem by using the highly efficient insertion and deletion procedures of the B$^+$-tree. The B^x-tree [4] is the first technique to adopt the B$^+$-tree to index moving objects. The B^{dual}-tree [3] and ST^{2}B-tree [5] are two recent improvements on the B^x-tree. Park et al. [23] propose an adaptive index management system for answering future queries. The index structure used is the Bst-tree [24] which incorporates the features of the B-tree and the multi-version tree [25, 26].

Although B$^+$-tree based spatiotemporal indexes are good for update performance, they suffer from low query performance. This is due to the reduction in dimensionality resulting from the need to enlarge queried regions in order to ensure no objects belonging to the result set are missed.

3. THE BENEFITS OF SHARED QUERY EXECUTION

In this section, we will describe how shared query execution can reduce I/O costs of queries by increasing the temporal locality of the reference stream and reducing sensitivity to overlapping MBRs.

The high level idea behind shared execution is to process a batch of queries concurrently by doing a single traversal of the TPR*-tree, visiting every node referenced by the union of the batch of queries.

3.1. Increased temporal locality

Buffers work best when there is high temporal locality in the reference stream. This is because higher temporal locality

means a smaller buffer is needed to avoid buffer misses for the same reference stream. Often, the amount of temporal locality present is a property of the data and workload and therefore, beyond our control. However, when we process queries in a batch, we have the opportunity to influence the amount of temporal locality present by changing the order in which operations are done. In this section, we compare the amount of temporal locality present when we process queries in arrival order, using shared query execution and in spatial proximity order.

We define shared execution as follows.

DEFINITION 3.1 (Shared Execution). *Given a set of operations O to perform on a tree, shared execution performs all the operations in O while producing a reference stream on the nodes of the tree such that every tree node is referenced at most once and the set of distinct nodes accessed equals the union of the nodes that would be accessed if the operations of O were performed individually.*

The above definition of shared execution can be applied to any of the following operations: query, insertion and deletion to produce shared query execution, shared insertion execution and shared deletion execution, respectively.

High temporal locality is achieved when the buffer miss rate decreases rapidly with increased buffer size. Therefore, we provide a definition for buffer miss rate as a function of buffer size. For a given input reference stream $S = \langle s_1, s_2, .., s_i, ..s_n \rangle$, where s_i is the page referenced at the i^{th} reference, buffer size M specified in pages, buffer replacement policy B, we define buffer miss rate as follows:

$$MissRate(S, M, B) = \frac{\sum_{s \in S} PageLoad(s, M, B)}{|S|} \quad (1)$$

where $PageLoad(s, M, B)$ is a function that returns 1 if reference s causes a page load, else it returns 0.

In an extreme case when M equals 1 page, the only way to avoid a page load (buffer miss) is to reference *the* page currently in the buffer. This is what leads us to propose the following shared query execution function ($SQF(S)$) which reorders the reference stream S to minimize the buffer miss rate while also meeting the shared execution definition (Definition 3.1). $SQF(S)$ is defined as follows:

DEFINITION 3.2 (Shared Query execution Function ($SQF(S)$)). *Given an input reference stream S, $SQF(S)$ outputs a reordered reference stream in which every reference to the same page is grouped to occur consecutively.*

This definition of SQF seems difficult to achieve since it seems to require knowledge of the complete set of pages references at the beginning. However, this knowledge is not required since we can produce this reference order by traversing the tree in a particular way, as shown in more detail in Algorithm 2 for the range query and Algorithm 3 for the k nearest neighbour query.

We next define a Lemma which states that in a situation where there is a buffer of size 1 page, $SQF(S)$ reorders the input stream to achieve minimum buffer miss rate.

LEMMA 3.1 (Maximum miss rate for RAM buffer size 1). *Given a reference stream S, a 1-page buffer and any buffer replacement algorithm B, assuming the buffer starts empty, the miss rate $MissRate(SQF(S), 1, B)$ is minimized when the reference stream is reordered using the shared query execution function $SQF(S)$.*

Proof. The number of distinct pages in the output stream of $SQF(S)$ equals $|\{s_i \neq s_{i-1} | \forall s_i \in SQF(S) \wedge 1 < i \leq n\}| + 1$ because $SQF(S)$ groups all references to the same page consecutively. Only references $s_i \neq s_{i-1}$ and s_1 require a page load because in the remaining situations ($s_j = s_{j-1}$), the page s_j can be found in the 1-page buffer. Therefore, the number of page loads by $SQF(S)$ is exactly equal to the number of distinct pages in S. The least number of pages that can be loaded for a given stream S is equal to the number of distinct pages in S. This is because the buffer starts empty and therefore we cannot avoid loading every distinct page at least once. Therefore, $SQF(S)$ achieves the lowest number of page loads possible and therefore the minimum miss rate. □

Using Lemma 3.1, we arrive at Theorem 3.1 which effectively says if we use the shared query execution function to reorder the reference stream, we can achieve the minimum miss rate using a buffer of just one page. Increasing the buffer size will not further decrease the miss rate.

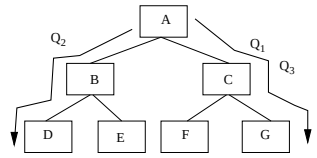
THEOREM 3.1 (Maximum miss rate for any buffer size). *Given a reference stream S, any buffer size $M > 0$, and any buffer replacement algorithm B, an empty initial buffer, the minimum miss rate $MIN_{p \in \pi(S)} MissRate(p, M, B) = MissRate(SQF(S), 1, B))$, where $\pi(S)$ is the set of all possible permutations of the reference stream S.*

Proof. This proof is a trivial extension of the proof for Lemma 3.1. As stated in the proof for Lemma 3.1, the number of page loads for $SQF(S)$ is equal to the number of distinct pages in S. No matter how much bigger we make the buffer, we cannot improve on this since we start with an empty buffer and we therefore need to load every distinct page into the buffer at least once. □

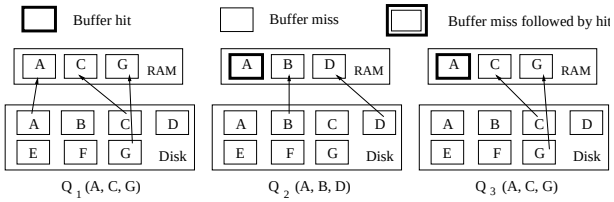
Our shared query execution algorithm produces an output reference stream which conforms to SQF and therefore, according to Theorem 3.1, it achieves the minimum possible miss rate with a read buffer of just one page. This frees the remainder of the buffer to buffer object insertions and deletions. In our system, a memory-based index exists which is used at query time concurrently with the disk-based index (more details in Section 4).

Note when the theorem is applied to our shared query execution, we minimize the intra-batch miss rate instead of the inter-batch miss rate. That is, we consider the reference stream S as the reference stream generated from executing the queries of the current batch instead of queries across separate batches.

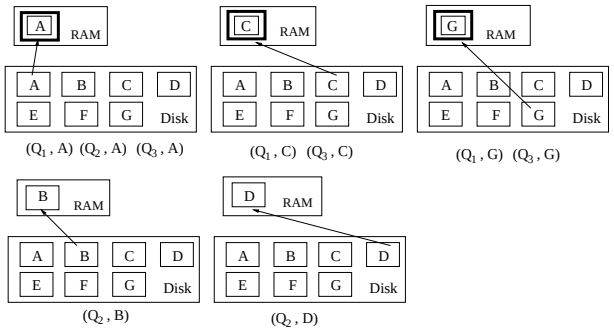
Figure 2 uses an example to illustrate the miss rate for processing three queries in arrival order against using shared



(a) Tree traversal for queries Q_1, Q_2 and Q_3



(b) Arrival Ordered Query Execution (Buffer size 3)



(c) Shared Query Execution (Buffer size 1)

FIGURE 2. Example of tree traversal in arrival order versus shared execution. Queries arrive in the following order: Q_1, Q_2 and Q_3.

execution. In this example, we assume the popular least recently used buffer replacement (LRU) policy is being used. Figure 2(a) shows the traversal path of the three queries Q_1, Q_2, and Q_3. Figure 2(b) shows the buffer hits and misses for processing the queries in the arrival order of $\langle Q_1, Q_2, Q_3 \rangle$ with a buffer size of 3. The example shows the three queries are processed with 7 buffer misses and therefore 7 page loads. Although there are 9 references, only 2 of them were buffer hits, because of the low temporal locality in the reference stream. This contrasts with shared execution, shown in Figure 2(c), where all the queries process the same node of the tree at the same time. The example shows, using shared execution with a buffer of just *1* page, we can achieve 4 buffer hits (2 hits on page A, 1 on C and 1 on G) and therefore only 5 buffer misses. 5 buffer misses is the minimum possible, since the three queries access 5 distinct pages (A, C, G, B and D). This is an illustration of Theorem 3.1.

3.2. Reduced sensitivity to overlap

As mentioned at the end of Section 2.1.1, another benefit of shared query execution is reduced sensitivity to MBR overlap. This is due to the fact that using shared query execution, the number of page loads equals exactly the number of distinct pages in the reference stream (as explained in Section 3.1). Therefore, if at least one query among the batch uses a page, then that page will be loaded exactly once, no matter how many queries touch that same page.

Figure 3 shows an example of two queries Q_1 and Q_2 traversing a tree with non-overlapping MBRs F and G (Figure 3(a)) and overlapping MBRs F and G (Figure 3(b)). Figure 3(c) and 3(d) compare the number of buffer misses for the non-overlapping and overlapping cases, respectively, when using shared query execution. The results show, in both cases, the number of buffer misses equals 4. The reason for the same number of buffer misses, despite the absence and presence of overlapping MBRs, is that the number of distinct page references is the same for both cases.

It is important to note that although shared execution reduces the impact of MBR overlap, it is still desirable for the MBR to have less overlap. We can think of it as follows: shared execution effectively works like creating one large query which is the union of the entire batch of queries. This large query will touch a lot of MBRs due to its large size, however, if there is more overlap between the MBRs, it will likely touch more MBRs than if there were less overlap.

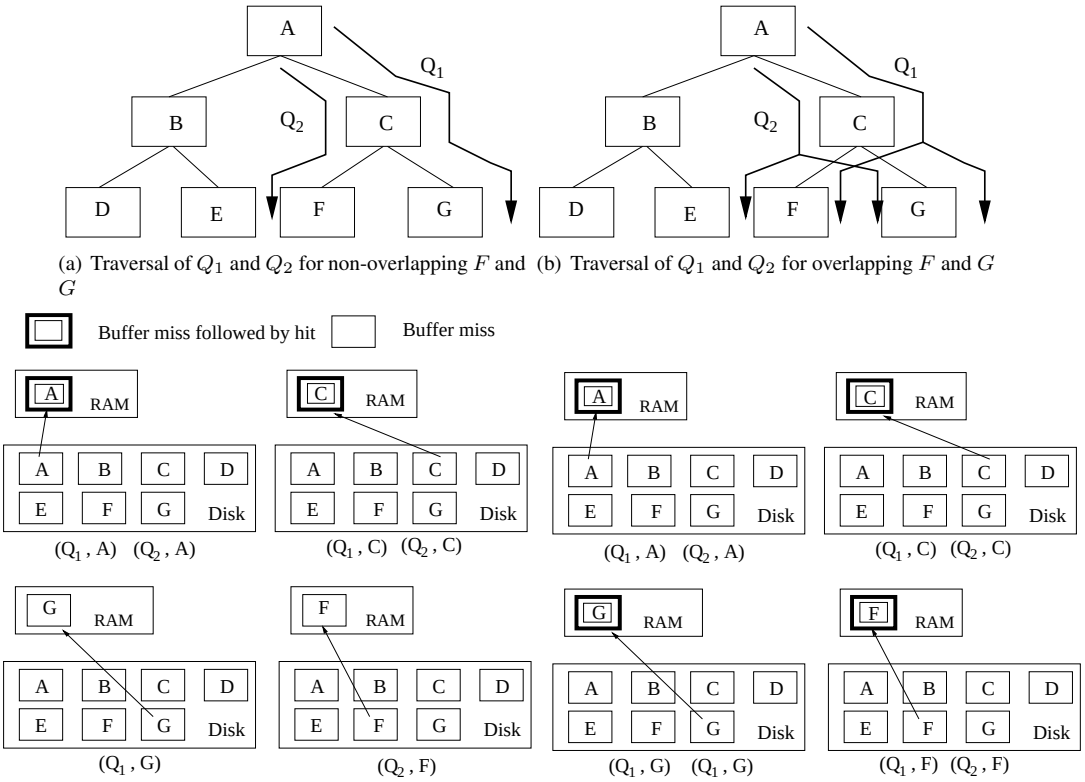
4. BUFFER-EFFICIENT SPATIOTEMPORAL INDEXING

In this section, we describe our S^{e}TPR*-tree which uses a combination of four techniques: dynamic RAM buffer allocation (DBufferAlloc); shared query execution; shared deletion execution; and proximity ordered insertion. These techniques are combined to minimize query and update I/O simultaneously. Although this section presents our ideas when applied to the TPR*-tree, they can be equally applied to the TPR-tree.

All four techniques are designed to obtain the maximum benefit from the use of the limited sized RAM buffer. Figure 4 shows a high level diagram of how our S^{e}TPR*-tree works. DBufferAlloc breaks up the buffer usage into two phases: the operation phase; and the merge phase. The operation phase caches updates and performs shared query execution. The merge phase merges the cached objects with the disk-based TPR*-tree.

DBufferAlloc assigns RAM buffer space to index operations dynamically, according to need. During the operation phase, we only allocate a 1-page buffer for answering queries. This is because we use shared query execution to execute batches of queries at once and thereby reduce the number of read I/O to the minimum possible with a read RAM buffer size of just *1* page (proved in Section 4.1). DBufferAlloc allocates the remaining RAM buffer to a RAM-based TPR*-tree which stores inserted objects. This reduces update costs by allowing us to delay updates to a later time. Deletions are also delayed in RAM and cached using a hash table for fast lookup. The higher resultant query I/O costs from delayed insertions and deletions is overcome by shared query execution.

When insertions and deletions have filled up the RAM buffer entirely, DBufferAlloc switches into the merge



(a) Traversal of Q_1 and Q_2 for non-overlapping F and G

(b) Traversal of Q_1 and Q_2 for overlapping F and G

(c) Shared execution of Q_1 and Q_2 for non-overlapping F and G

(d) Shared execution of Q_1 and Q_2 for overlapping F and G

FIGURE 3. Example showing the insensitivity of shared query execution to overlapping MBRs.

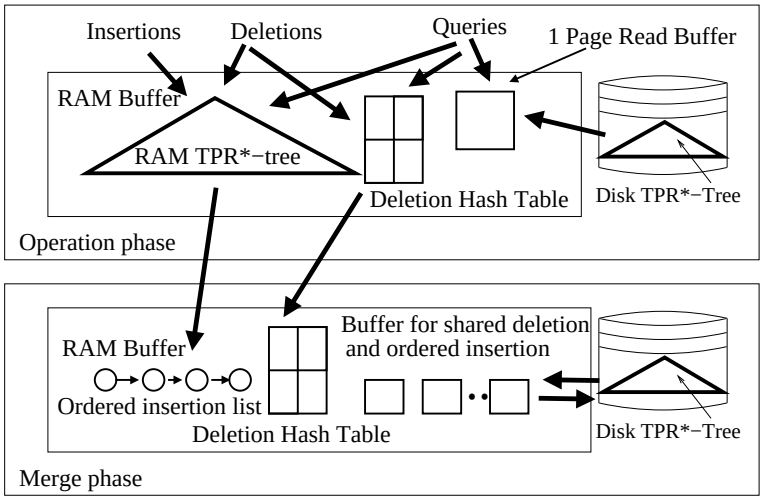


FIGURE 4. Diagram illustrating operations on the S^{e}TPR*-tree

phase and reallocates the RAM buffer space as follows. DBufferAlloc converts the RAM-based TPR*-tree into a proximity ordered list of objects, thereby freeing RAM which is then used for a multi-page deletion/insertion buffer. We then delete all the objects in the deletion hash table from the disk-based TPR*-tree in bulk via shared deletion execution. This increases the temporal locality which results in making more efficient use of the deletion/insertion buffer. Finally, we insert all the objects in the RAM-based TPR*-tree into the disk-based TPR*-tree in proximity order (benefits explained in Section 4.4). This also increases the temporal locality of the insertions which results in more optimized insertion buffer usage. At the end of the merge phase, the operation phase is restarted.

Algorithm 1 gives the high level algorithm for our S^{e}TPR*-tree. In the merge phase (Lines 10 - 16), we perform deletion and insertion separately because shared insertion execution would lead to a less optimized tree (see Section 4.4 for an example). Therefore, instead of shared insertion execution, we do proximity ordered insertion for

Algorithm 1: High Level Algorithm(o)

```
   Input: Operation o
1  if operation o is batched range query then
2      invoke Shared Query Execution on RAM-based TPR*-tree
3      invoke Shared Query Execution on disk-based TPR*-tree
4      combine results from querying both trees
5  else if operation o is batched kNN query then
6      invoke Shared kNN Query Execution on RAM-based TPR*-tree
7      invoke Shared kNN Query Execution on disk-based TPR*-tree
8      combine results from querying both trees
9  else
10     if RAM buffer is full then
11         // execute merge phase
12         move objects from RAM-based TPR*-tree to object insertion list
13         enlarge disk-based TPR*-tree's buffers to occupy free buffer space
14         invoke Shared Deletion Execution using objects in deletion hash table
15         invoke Proximity Ordered Insertion using objects in insertion list
16         flush all pages in disk-based TPR*-tree's buffer and reset buffer size to 1 page
17     if operation o is insertion then
18         insert object into RAM-based TPR*-tree
19     else if operation o is deletion then
20         if object exists in RAM-based TPR*-tree then
21             delete object from RAM-based TPR*-tree
22         else
23             insert object into deletion hash table
```

the TPR*-tree. However, as our experiments in Section 6.2 show, our proximity ordered insertion which has high temporal locality reduces I/O costs significantly compared to arrival ordered insertion. Sections 4.3 and 4.4 describe our shared deletion execution and proximity ordered insertion algorithms, respectively.

In batched query execution (either range or kNN queries) (Lines 1 - 8), we first perform shared query execution on the RAM-based TPR*-tree because it contains all the most recently inserted objects. The reason we perform shared query execution on the RAM-based TPR*-tree is because shared query execution results in higher temporal locality which results in less level 1 and 2 CPU cache thrashing. For the RAM-based TPR*-tree, we found a node size of 512 bytes gave best performance. When performing shared execution on the disk-based TPR*-tree, we do not check whether the same object was found in the RAM-based TPR*-tree since any object that exists in both must also be in the deletion hash table. When using the disk-based TPR*-tree, we remove from the result any objects which are in the deletion hash table. Sections 4.1 and 4.2 present the algorithms used for shared query execution for range queries and k nearest neighbour queries, respectively.

It is important to note that one limitation of shared query execution is that it cannot be used when there are certain types of dependencies between queries, for example, if the parameters of one query depend on the results of an another query.

4.1. Shared range query execution

Algorithm 2: Shared Range Query Execution(QL, DL)

```
   Input: range query list QL, deletion list DL
   Output: result set RS⟨query, entry⟩
1  // let LE be a list of node entries that need to be processed
2  initialize an empty LE
3  insert root entry to LE
4  while LE is not empty do
5      remove the last entry le from LE
6      load the node N pointed by le
7      initialize an empty Query List QL'
8      insert all queries from QL which intersect with N to QL'
9      if N is a leaf node then
10         for each entry e ∈ N do
11             if querying RAM-based tree or e ∉ DL then
12                 if e intersects with query q ∈ QL' then
13                     insert a pair ⟨q, e⟩ into RS
14     else
15         /* N is a non leaf node */
16         insert all entries ∈ N which intersect with any query in QL' into LE
17 return RS
```

In this section, we describe our algorithm for shared range query execution (Algorithm 2). The algorithm works by doing a depth-first traversal down the TPR*-tree with an initial list of queries. The tree traversal only follows down children entries if at least one of the queries in the query list intersects with the children node. Therefore, the pruning ability of the TPR*-tree is maintained. As we traverse down the tree, the query list QL is trimmed down by using QL' which only stores the queries that intersect the current node N. We next prove that Algorithm 2 is correct.

THEOREM 4.1 (Algorithm 2 is correct). *Algorithm 2 returns for each of the range queries in the query list QL, the complete set of intersecting valid (non-deleted) objects.*

Proof. Every node that intersects at least one of the list of range queries is traversed. This can be seen from Line 16.

For every leaf node traversed, all of its entries (objects), which intersect at least one of the range queries which have not been deleted, are inserted into the result set of the corresponding query. This can be seen from Lines 9 to 13. Therefore, for each of the range queries in the query list QL, the complete set of intersecting valid (non-deleted) objects are returned. □

We next prove that Algorithm 2 performs shared query execution according to Definition 3.2:

THEOREM 4.2 (Algorithm 2 conforms to SQF). *Algorithm 2 reorders the index node reference stream of the batch of queries to conform to the definition of the output of the shared query execution function (SQF).*

Proof. We need to prove two facts. First, Algorithm 2 accesses the same set of nodes as arrival order processing. Second, the sequence of node accesses by Algorithm 2 has the same property as the output of SQF, namely all references to the same page (in this case index nodes) are accessed consecutively. The first fact is true since Line 16 ensures that the traversal only loads nodes that intersect with at least one query in QL'. Therefore, a node that does not intersect any query is never loaded. The second fact is true since Algorithm 2 does a single pruned depth first traversal which never revisits a node that has already been visited. □

By combining Theorem 4.2 and 3.1, we can conclude Algorithm 2 achieves the minimum possible miss rate given a RAM buffer of size one page.

It is important to note that Algorithm 2 trades extra computation costs for reduced temporary memory usage. This is because it uses a none pruned QL in Line 8 in each pass of the while loop. Therefore, we need to look through the entire QL every time we create QL'. The benefit of this approach is that we do not need to keep a temporary pruned query list for each node traversed but not yet finished processing. These temporary query lists can occupy a lot of memory if the query batch size and tree size are both large.

4.2. Shared k nearest neighbour query execution

Our shared k nearest neighbour (kNN) query execution (Algorithm 3) works by doing a shared depth-first traversal down the TPR*-tree. The tree traversal works the same as the non-shared execution kNN query traversal algorithm of the TPR*-tree, except we simultaneously perform the traversal for the entire batch queries at the same time. We keep track of the traversal meta-data for each query in the batch in the SLE_q list and the result set data structure RS_q. The algorithm, like the non-shared execution kNN query traversal algorithm, traverses down to the entry at the top of the SLE_q list first (Line 7). This corresponds to the entry which has the minimum distance to the query point. We do not care which of the SLE_q lists is used first to drive the traversal because we know each entry at the top of every SLE_q must be traversed sooner or later.

When we traverse down an entry, we update the meta-data for all queries. If the node pointed to by the entry is a leaf node, then we update the current kNN result set (RS_q) for any query q which contains at least one object in RS_q which could potentially be further from q than the objects in the currently considered leaf node (Line 18). If the node is a non-leaf node, then we insert each of its children which could potentially contain an entry that contains one or more objects belonging to its kNN into its SLE_q list (Lines 21 - 23). We next prove that Algorithm 3 is correct.

Algorithm 3: Shared kNN Query Execution(QL, DL)

Input: kNN query list QL, deletion list DL
Output: set of results $RS_q\langle rs, max_{dist}\rangle$ for each query $q \in QL$, where rs is the set of kNN objects to q and their associated distance to q and max_{dist} is the maximum distance of all objects in rs

1 // let $SLE_q\langle entry, dist\rangle$ be a list of node entries sorted by distance ($dist$) associated with a kNN query q
2 **initialize** the SLE_q for each query $q \in QL$ to contain the root entry
3 **initialize** RS_{kNN} to empty
4 // let $\langle le, dist_{q\to le}\rangle$ be the current node entry (le) and the distance from q to le ($dist_{q\to le}$)
5 **while** QL *is not empty* **do**
6 **for** *each query* $q \in QL$ **do**
7 $\langle le, dist_{q\to le}\rangle \leftarrow$ **remove** the last entry from SLE_q
8 **if** $|RS_q.rs| < q.k$ || $dist_{q\to le} < RS_q.max_{dist}$ **then**
9 **for** *each query* $q' \in QL \backslash \{q\}$ **do**
10 **remove** le from $SLE'_{q'}$
11 **exit** the **for** loop (Line 6)
12 **else**
13 **remove** q from QL // q is completed
14 **if** QL *is empty* **then**
15 **return** set of all results
16 **load** the node N pointed by le
17 **if** N *is a leaf node* **then**
18 **invoke Shared** k**NN Search Leaf Execution** (N, QL, RS_{kNN}, DL) (Algorithm 4)
19 **else**
20 // N is a non-leaf node
21 **for** *each query* $q \in QL$ **do**
22 **for** *each entry* $e \in N$ **do**
23 **if** $|RS_q.rs| < q.k$ || $dist_{q\to e} < RS_q.max_{dist}$ **then**
24 insert $\langle e, dist_{q\to e}\rangle$ into SLE_q

THEOREM 4.3 (Algorithm 3 is correct). *Algorithm 3 returns for each kNN query in the query list QL, the set of k valid (non-deleted) objects which have closest distance to the query.*

Algorithm 4: Shared kNN Search Leaf Execution(N, QL, RS_{kNN}, DL)

```
Input: leaf node N, kNN query list QL, kNN result set
       RS_kNN, deletion list DL
1  initialize an empty query list QL'
2  for each query q ∈ QL do
3      if |RS_q.rs| < q.k || dist_{q→N} < RS_q.max_dist then
4          insert q into QL'
5  for each query q ∈ QL' do
6      for each entry e ∈ N do
7          if querying RAM-based tree or e ∉ DL then
8              if |RS_q.rs| < q.k then
9                  insert ⟨e, dist_{q→e}⟩ into RS_q.rs
10                 if dist_{q→e} > RS_q.max_dist then
11                     RS_q.max_dist ← dist_{q→e}
12             else
13                 if dist_{q→e} < RS_q.max_dist then
14                     replace ⟨e', RS_q.max_dist⟩ ∈
                       RS_q.RS by ⟨e, dist_{q→e}⟩
```

Proof. Algorithm 3 performs a top down traversal visiting all nodes containing the k nearest objects. Every branch of the tree is traversed except for branches which is pruned by the pruning rule. The pruning rule avoids traversing those nodes whose distance from every query $q \in QL$ is greater than the current maximum distance of the current k nearest objects of the corresponding query or the query has less than k nearest objects found so far. If a node satisfies the pruning rule it can not contain any of the k nearest objects for any query. This is because it means all the space that the pruned node covers are further than the maximum distance of the k nearest objects founds so far for every query. Lines 6 - 13 and 21 - 24 of Algorithm 3 and Lines 2 - 4 of Algorithm 4 show the application of the pruning rule during the top down traversal of the tree. For every leaf node traversed Lines 5 - 14 of Algorithm 4 updates the current k nearest objects of every query by doing a linear scan through all the objects of the traversed leaf node and comparing the distance of each scanned object to the k nearest objects of every query. Therefore, for each kNN query in the query list QL, the complete set of k nearest objects is returned. □

We next prove that Algorithm 3 performs shared query execution according to Definition 3.2:

THEOREM 4.4 (Algorithm 3 conforms to SQF). *Algorithm 3 reorders the index node reference stream of the batch of queries to conform to the definition of the output of the shared query execution function (SQF).*

Proof. We need to prove two facts. First, Algorithm 3 accesses the same set of nodes as arrival order processing. Second, the sequence of node accessed by Algorithm 3 has the same property as the output of SQF, namely all references to the same page (in this case index nodes) are accessed consecutively. The first fact is true since it always traverses down the nodes at the top of the SLE_q lists first (Line 7). This is the same as the traversal pattern of arrival order processing. The second fact is true since Algorithm 3 does a single pruned depth-first traversal which never revisits a node that has already been visited, since it updates the meta-data for all the queries as soon as an entry is visited. □

By combining Theorem 4.4 and 3.1, we can conclude Algorithm 3 achieves the minimum possible miss rate given a RAM buffer of size one page.

4.3. Shared deletion execution

Algorithm 5: Shared Deletion Execution(DL)

```
Input: object deletion list DL
1  initialize an empty reinsertion list L_reinsert
2  load a root node N
3  invoke Shared Node Deletion Execution(N, DL,
   L_reinsert)
4  for each entry e in the L_reinsert do
5      invoke Insert(e) /* same as TPR*-tree's insertion
       algorithm */
```

In this section, we describe our algorithm for shared deletion execution (Algorithm 5). The algorithm first deletes all objects in the object deletion list DL, using shared deletion execution (see Algorithm 6). During the shared deletion execution, some nodes underflow. The children of these nodes are placed in the list $L_{reinsert}$, which are then reinserted into the disk-based TPR*-tree in Line 5. These objects are inserted using proximity ordered insertion since Algorithm 6 places the nodes that need reinsertion in the depth-first traversal order of the disk-based TPR*-tree. Again, we use proximity ordered insertion instead of shared insertion execution in order to keep the tree more optimized (see the example in Section 4.4).

Algorithm 6 shows the shared deletion algorithm used to delete the objects in the deletion list DL from the disk-based TPR*-tree. The algorithm works by doing a depth-first traversal of the tree, using shared execution. During the execution of this algorithm, we use the most recently used buffer replacement policy which resembles the replacement strategy of a stack. This makes sense because after visiting children nodes, the parent node sometimes needs to be accessed again to tighten the MBR of the child node. The algorithm performs a shared traversal of the tree, thereby reducing the high read I/O costs of performing many individual deletions. It only traverses down paths which intersect at least one object in the deletion list DL (Lines 11 - 14), thereby using the tree to prune the traversal. The algorithm also places the children of underflow non-leaf nodes (Line 25) and objects of underflow leaf nodes (Line 4) into the reinsertion list $L_{reinsert}$. These objects are later reinserted into the disk-based TPR*-tree.

Algorithm 6: Shared Node Deletion Execution(N, DL, $L_{reinsert}$)

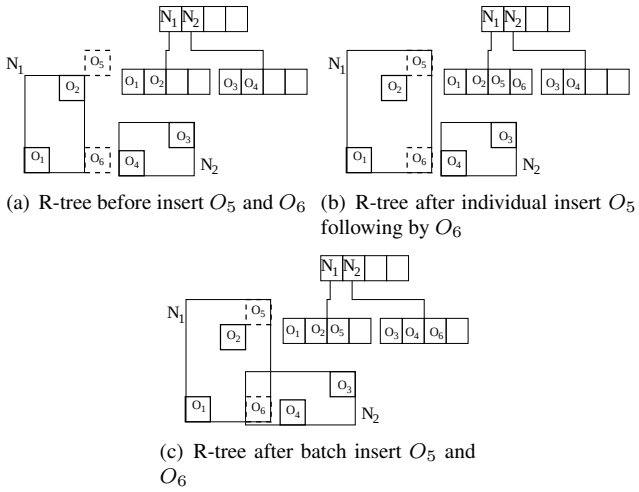
```
Input: node N, object deletion list DL, reinsert list L_reinsert
1  if N is leaf node then
2      remove from both N and DL all the objects which are in both containers
3      if N underflows then
4          copy all remaining entries of N to L_reinsert
5          mark N as an obsolete node, to be deleted
6  else
7      // N is a non leaf node
8      initialize an empty list CL_deleted which holds array index to deleted children entries
9      for each entry e in N do
10         initialize an empty current object deletion list DL'
11         insert all objects from DL which intersect with N into DL'
12         if DL' is not empty then
13             load a child node N' pointed by e
14             invoke Shared Node Deletion Execution (N', DL', L_reinsert)
15             if N' marked as an obsolete node then
16                 insert index of e into CL_deleted
17             else
18                 tighten MBR of e according to TPR*-tree heuristics
19             remove all objects from DL which is no longer in DL' after the recursive call
20             if DL is empty then
21                 // terminate search because all objects have been deleted
22                 exit the for loop (Line 9)
23     remove all entries in CL_deleted from N
24     if N underflows then
25         copy all remaining entries of N into L_reinsert
26         mark N as an obsolete node, to be deleted
```

The shared deletion algorithm cannot reorder the reference stream to conform to the output of the SQF and hence get minimum miss rate with a buffer size of 1 page. This is because it needs to tighten the MBR of the parent node *after* traversing down to the child. However, given a buffer size equal to the height of the tree, we can avoid loading any page more than once. This is proved in the following theorem.

THEOREM 4.5 (Shared Deletion Execution). *Given a buffer with size equal to the height of the index tree M_h which is evicted using the most recently used replacement policy (like a stack), Algorithm 6 never loads a page (tree node) more than once.*

Proof. The maximum depth of the recursion of Algorithm 6 is M_h. This is because Line 14 is the only place the function recursively calls itself and it uses N' as its argument which is a child of the current non-leaf node N. Lines 16 and 18 are the only places a previously loaded node (page) are reused. Both of these references are to entries of the current node N (Line 12) whose child was just traversed in the recursive call on Line 14. Once we come out of a recursion, we never reference a node that was just recursed down to. Therefore, as long as the buffer can fit M_h pages, we can safely evict the most recently accessed page without ever needing it again. □

4.4. Proximity ordered insertion



(a) R-tree before insert O_5 and O_6 (b) R-tree after individual insert O_5 following by O_6

(c) R-tree after batch insert O_5 and O_6

FIGURE 5. Example of how shared execution insertion can cause more MBR overlap than individual insertion

In this section, we describe our simple proximity ordered insertion algorithm and use an example to show why shared insertion execution would lead to a less optimized tree compared to individual insertion.

Proximity ordered insertion improves cache locality because objects that are close to each other often share the same or similar traversal paths. Therefore, when nearby objects are inserted in succession, there is a high probability of cache hits. Our proximity ordered insertion works as follows. We first traverse the RAM-based TPR*-tree in depth-first order and insert the objects encountered into an insertion list in traversal order. The order of the objects in the insertion list has proximity order since the nodes of the TPR*-tree groups objects in proximity order. This method is faster than using methods such as the Peano curve or Hilbert curve, since the TPR*-tree has already laid out the objects in proximity order for us. We then insert the objects in that order into the disk-based TPR*-tree.

Figure 5 shows an example that demonstrates why shared insertion execution can create a less optimized tree (more MBR overlap) compared to individual insertion. Figure 5(a) shows the set of objects O_1, O_2, O_3 and O_4 in the tree and the two objects O_5 and O_6 that need to be inserted. Figure

Parameter	Setting
Space domain	**100,000x100,000m^2**
Query type	range query, kNN query
Data size (objects)	**100K**, ..., 1M
Maximum object speed	10m/ts, ..., **100m/ts**
Maximum update interval	**120ts**
Update frequency	**1**, ..., 10
Range query side length (m)	2,000, **10,000**, ..., 20,000
Number of neighbours, k	100, ..., 1000
Query batch size	1, 5, 10, **25**, 50, 100
Query predictive time	**60ts**
Time duration	**240ts**, 600ts
Buffer size (pages)	**50**, 4, 8, 16, 32, ..., 256
Disk page size	**4KB**
Update/query ratio	**1:100**

TABLE 1. Parameters and Their Settings. Note ts is an abbreviation for time stamp.

5(b) shows the resultant tree after object O_5 and O_6 have been inserted individually. Assume the order of insertion is O_5 and O_6. O_5 will be inserted into leaf node N_1 since this results in the least amount of node expansion. Next, when O_6 is inserted, it can be placed directly into N_1 with no further need for node expansion. The final result is that the two nodes N_1 and N_2 do not overlap. In contrast, consider O_5 and O_6 are inserted using shared insertion execution. In this case, O_5 and O_6 may be inserted into different nodes, as shown in Figure 5(c). The reason O_6 is inserted into N_2 instead of the more optimal N_1 is that in shared insertion execution, the search for the best node to insert into would be done together for all the nodes at the same time. In such a case, the only computationally feasible way to determine the best node each object should be inserted into is to consider the insertions independently, assuming no other objects have been inserted yet. In that case, inserting O_6 into N_2 will be considered better than N_1 since it results in less node expansion, assuming O_5 has not yet been inserted.

5. EXPERIMENTAL SETUP

In this section, we first describe the benchmark used in our experiments in Section 5.1. Next, in Section 5.2, the rival algorithms used in our experimental study are described. Section 5.3 describes the metrics we used to measure the algorithms. Finally, in Section 5.4, we describe the hardware used to test the algorithms.

5.1. Benchmark setup

The experiments were conducted using the benchmark defined in Chen et al. [27] for evaluating moving object indexes. The data sets used in the experiments was the uniform data set generated by the benchmark's data generator which was downloaded from [28].

The parameters used in the experiments are summarized in table 1, where values in bold denote the default values used. For more details on the benchmark settings, please refer to [27].

5.2. Algorithm setup

We compare our S^{e}TPR*-tree against three rival state-of-the-art spatiotemporal indexes: the TPR*-tree, the B^x-tree and the RUM*-tree (a memo-based update approach applied to the TPR*-tree). We used the source code for the TPR*-tree, the B^x-tree and the RUM*-tree provided by Chen et al. from [28]. We modified the TPR*-tree code to make the S^{e}TPR*-tree. All code was implemented in C++ under Microsoft Visual C++ 2008 running on Windows XP Professional SP3. The algorithms compared are described as follows:

- **S^{e}TPR*-tree**. The S^{e}TPR*-tree was built on top of the TPR*-tree using the shared execution techniques described in Section 4. The RAM-based TPR*-tree has a node size of 512 bytes and the disk-based TPR*-tree has a node size 4KB (1 page).
- **TPR*-tree**. The TPR*-tree proposed by Tao et al. [2] is optimized for the range query with default size 10,000x10,000m^2.
- **B^x-tree**. The B^x-tree proposed by Jensen et al. [4] using the Hilbert curve with the grid order λ equals 8 for space partitioning. The B^x-tree has two partitions. The velocity histogram contains 1,000 x 1,000 cells.
- **RUM*-tree**. The RUM-tree [8] was modified to work on top of the TPR*-tree instead of the traditional R-tree. This is the same approach as that used in Chen et al. [27]. A total of 10 tokens was used where each token is passed to another leaf node after every 1,000 updates.

In addition to the above algorithms, we also compared the TPR-tree but found its performance was very similar to the TPR*-tree, so therefore we did not report its results.

5.3. Measurement metrics

Our experiments measure the following main metrics: average I/O per query; average I/O per update; average execution time per query; and average execution time per update. In Section 6.10, we also measure query performance in terms of throughput and response time. The execution time, throughput and response time results include both CPU and I/O time. Our experiments were conducted on the Windows XP operating system (OS), which automatically caches all I/O requests. This effectively invalidates our own RAM buffer since a miss on our buffer may become an OS cache hit. Therefore, we **disabled the operating system's caching functionality**. If we had used the OS caching functionality, all objects would be cached since we cannot make the RAM buffer smaller than the index size. In this case, all the algorithms would just populate the cache and then never load anything ever again. Therefore, we bypassed the OS buffer and used our own controlled buffer. All algorithms were allocated the same RAM buffer size. Our algorithms used the buffer management technique described in Section 4. The other algorithms used a LRU buffer. This setup is different from the benchmark of Chen et al. [27],

where the CPU time is reported instead of execution time. The reason is that we would like to study the efficiency of buffering on total execution time, including both CPU and I/O costs.

5.4. Hardware setup

All experiments were conducted on a PC powered by Intel Core i7 CPU 2.8GHz with 4GB DDR3 main memory, and using a 64GB G.Skill Solid State Drive FM-25S2S-64GB (SSD). By default for use the SSD rather than a traditional magnetic hard disks because SSDs are very efficient for random reads and writes [29] and the price of SSDs has come down significantly. However, we have also included an experiment (shown in Section 6.11) using a magnetic hard disk with the same default settings to evaluate all algorithms.

6. EXPERIMENTAL RESULTS

In this section, we report the results of experiments, illustrating the performance of our S^{e}TPR*-tree against the TPR*-tree, the B^x-tree and the RUM*-tree.

6.1. Effect of range query batch size

In the first set of the experiments, we varied the query batch size used by our S^{e}TPR*-tree from 1 to 100 queries. Figures 6(a) and (b) show that the query performance of the S^{e}TPR*-tree improves significantly as the number of queries per batch increases. This is because as the batch size increases, the S^{e}TPR*-tree is more effective at increasing the temporal locality of the page reference stream of the queries. All other indexes perform the same, regardless of the batch size because they execute queries individually and therefore cannot take advantage of large batch sizes. The S^{e}TPR*-tree outperforms all other indexes at a batch size of 5 for query I/O and it outperforms the rest at a batch size of 25 for execution time. This indicates the S^{e}TPR*-tree needs a larger batch size to outperform the other indexes for execution time compared to query I/O. This is due to a combination of two additional computational costs that the S^{e}TPR*-tree incurs. First, it needs to query both the RAM-based TPR*-tree and disk-based TPR*-tree, whereas the other indexes do not need to query a RAM-based tree. Second, to save temporary memory space, it does not fully utilize the pruning capability of the TPR*-tree to reduce computation costs, as explained at the end of Section 4.1. We observed that at batch size of 100, the S^{e}TPR*-tree outperforms its nearest competitor, the TPR*-tree, by a factor of 5.6 for query I/O and 3.7 for query execution time.

The seemingly strange result that the S^{e}TPR*-tree's query execution time performance is worse at a batch size of 5 compared to 1 can be explained as follows. When performing shared query execution, there is tradeoff between higher CPU cost versus lower I/O costs. At batch size 1, there is no CPU overhead associated with shared query execution. The lower number of I/O of batch size 5 compared to batch size 1 is not enough to offset the higher CPU overheads. When the batch size is greater than 5, the I/O saving begins to outweigh the CPU overheads.

The S^{e}TPR*-tree achieves its good performance by increasing temporal locality of page references. The effect of temporal locality on performance is largely dependent on RAM buffer size. Therefore, in Figure 7, we vary both query batch size and RAM buffer size. In this experiment, we only compared the S^{e}TPR*-tree with the TPR*-tree. The reason is that the TPR*-tree was found to give the best query performance compared to other competing indexes tested and we did not want to clutter the graph with too many curves. Section 6.7 shows the performance of the other competing indexes with varying RAM buffer size.

The results in Figure 7 show that the performance of the S^{e}TPR*-tree does not change with RAM buffer size for a given query batch size. The reason can be explained by Theorem 4.2 which effectively states our shared query execution algorithm only needs one RAM buffer page to achieve the same number of buffer misses as any sized RAM buffer. When the RAM buffer size is large, the S^{e}TPR*-tree needs to process larger query batches before it can outperform the TPR*-tree. This is because a large RAM buffer size is more forgiving of the low temporal locality of the TPR*-tree.

6.2. Benefits of shared deletion execution and proximity ordered insertion

In this experiment, we explore the individual performance advantages of using shared deletion execution (SD) and proximity ordered insertion (PI). We wanted to further explore if performing entire update operations (deletion followed by insertion of same object) together is better than deleting all objects followed by inserting all objects. So accordingly, we have tested the following insertion deletion combination approaches:

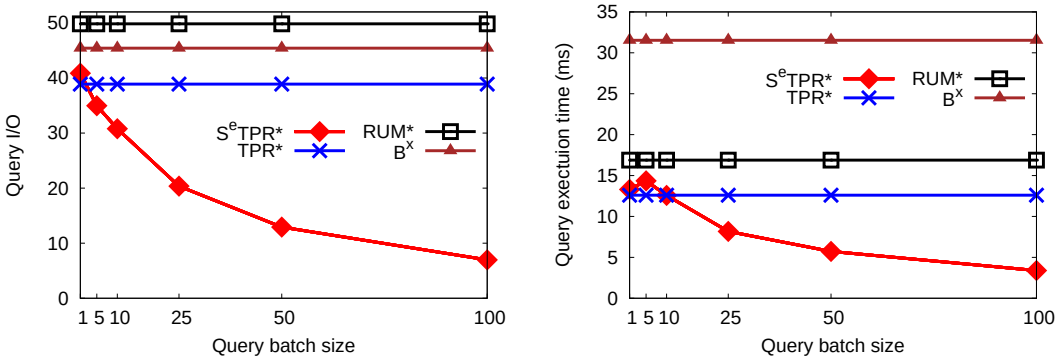
TPR*-tree approach: this is the traditional TPR*-tree approach which performs deletions and insertions in pairs in arrival ordered.

S^{e}TPR*-tree approach: this is our approach which uses both shared deletion execution and proximity ordered insertion techniques to perform updates. The S^{e}TPR*-tree first applies shared deletion execution to delete all objects in a batch, then inserts all objects in proximity order.

Naive 1 $\langle ID - AI \rangle$ approach: this approach performs updates by separating deletions and insertions. It deletes all objects individually in arrival order (ID) first and then inserts all objects individually in arrival order (AI).

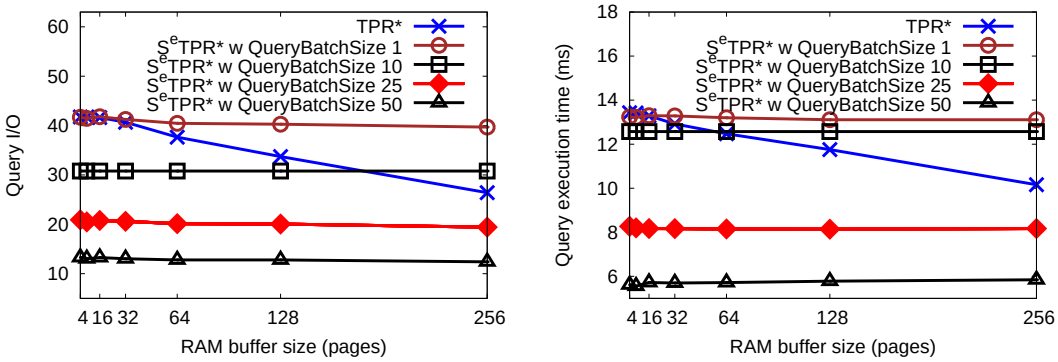
Naive 2 $\langle ID - PI \rangle$ approach: this approach also performs updates by separating deletions and insertions. It still uses normal individual deletions in arrival order (ID), but applies the proximity ordered insertion technique (PI).

Naive 3 $\langle SD - AI \rangle$ approach: this is the opposite of the Naive 2 $\langle ID - PI \rangle$ approach, namely it applies shared



(a) Query I/O (b) Query execution time

FIGURE 6. Effect of query batch size



(a) Query I/O (b) Query execution time

FIGURE 7. Effect of varying both query batch size and RAM buffer size

deletion execution (SD) followed by arrival ordered (AI) insertion.

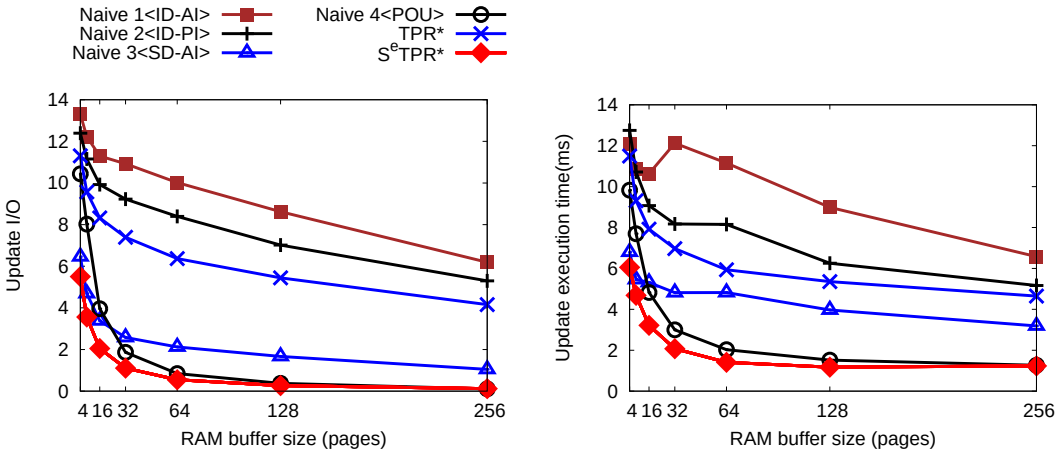
Naive 4 $\langle POU \rangle$ approach: this approach performs deletions and insertions in pairs, but performs these updates in proximity order (POU).

The results of this experiment are shown in Figure 8. The results show that the S^{e}TPR*-tree, which uses shared deletion execution and proximity ordered insertion, outperforms all the other approaches. This is because both techniques together achieve the highest temporal locality of page references. It is interesting to note that SD by itself without PI (Naive 3 $\langle SD - AI \rangle$ approach) already significantly outperforms the normal way updates are handled by the TPR*-tree. However, proximity ordered insertion by itself without shared deletion (Naive 2 $\langle ID - PI \rangle$ approach) performs worse than Naive 4 $\langle POU \rangle$ approach which performing updates according to their proximity ordered insertions. This indicates shared deletion execution has a larger positive impact on performance than proximity ordered insertion. However, when used together, they both contribute to improved performance. Finally, it is interesting to note that proximity ordered updates (Naive 4 $\langle POU \rangle$ approach) perform almost as well as using both shared deletion execution and proximity ordered insertion (S^{e}TPR*-tree approach) when buffer size is large.

6.3. Effect of data size

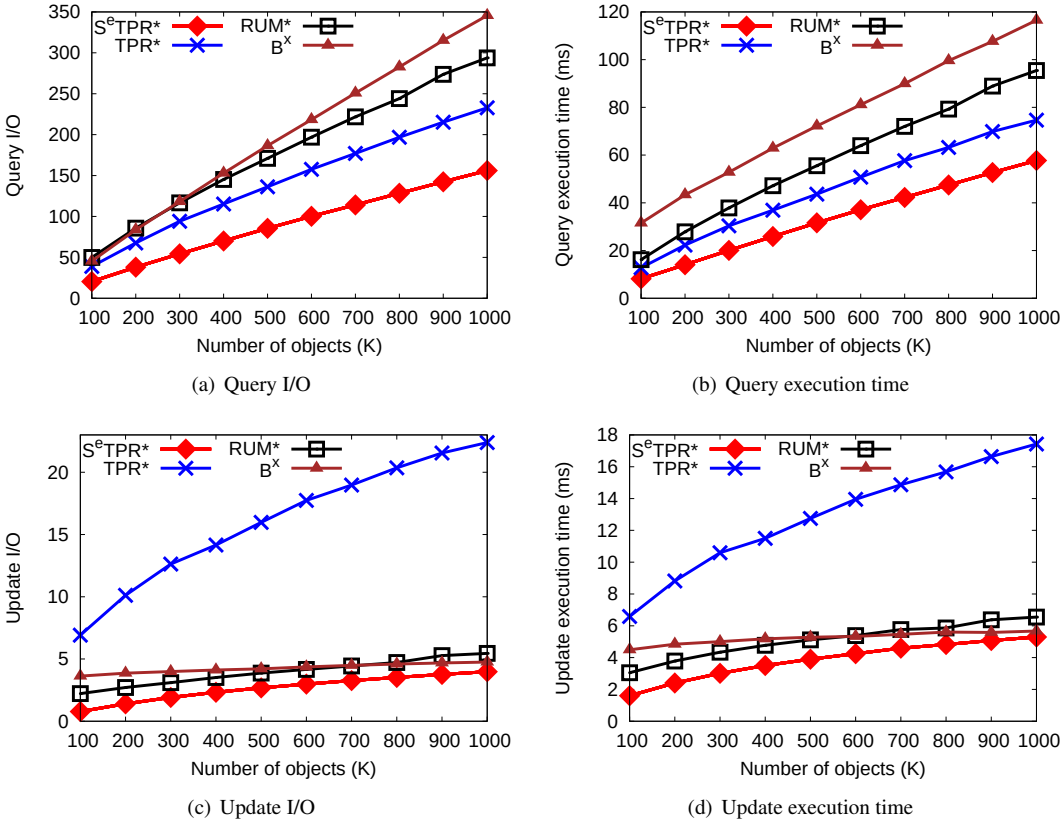
In this experiment, we examine the update and query performance of each index while varying the number of objects from 100K to 1M. As the data size grows, Figure 9 shows that the query performance decreases approximately linearly across all indexes. The results show that the S^{e}TPR*-tree has the best query performance. This is because of the shared query execution used by the S^{e}TPR*-tree. We observed that the B^x-tree has the worst query performance. This is due to its structure which requires it to expand the size of the queried region. The RUM*-tree's query performance is worse than the TPR*-tree due to the obsolete entries left in the tree. When the data set reaches 1M objects, the S^{e}TPR*-tree is 2.2 times better than the B^x-tree in terms of the average number of I/O per query.

The S^{e}TPR*-tree consistently outperforms the other indexes for update performance. Its update performance approaches that of the B^x-tree and the RUM*-tree as the number of objects approaches 1,000,000. The reason for this is that the buffer size stays the same in this experiment. Therefore, as the number of objects increases, our proximity ordered insertion begins to thrash the buffer since the buffer becomes too small to fit the working set. The TPR*-tree has



(a) Update I/O (b) Update execution time

FIGURE 8. Benefits of shared deletion execution and proximity ordered insertion



(a) Query I/O (b) Query execution time

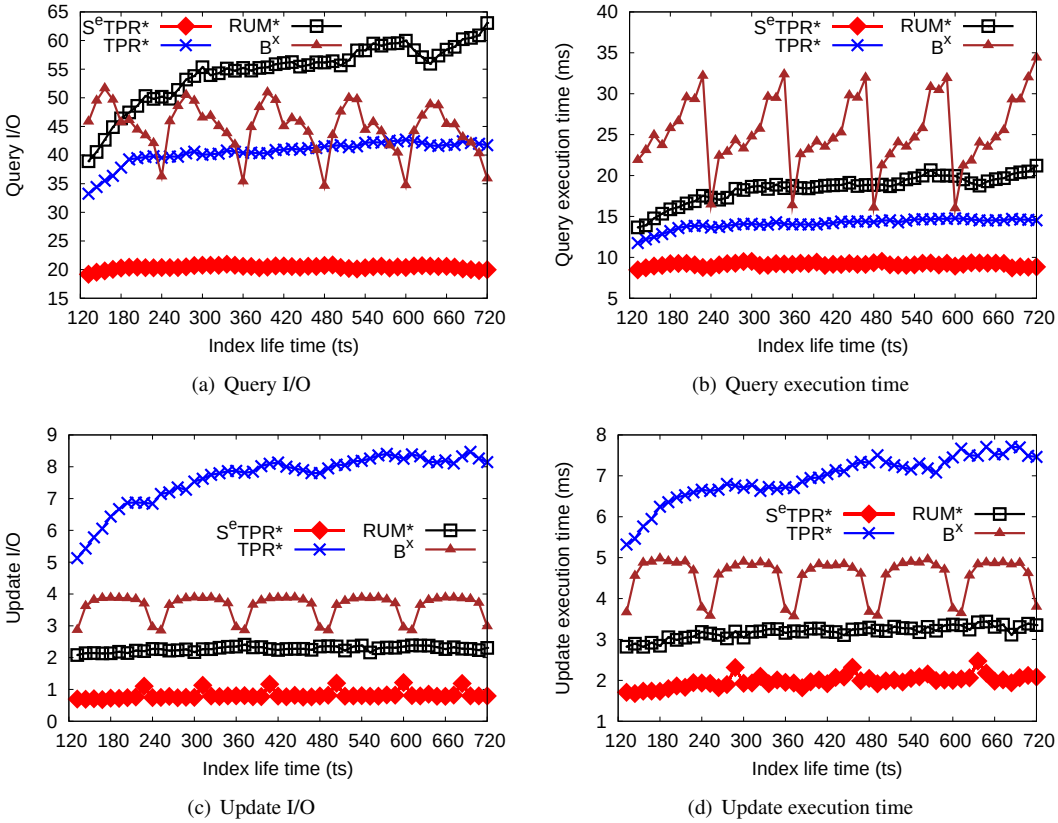
(c) Update I/O (d) Update execution time

FIGURE 9. Effect of data size

worst update performance since during object insertion and deletion, it traverses multiple paths when there is overlap MBRs. As the density of objects increases so does the amount of MBR overlap.

6.4. Effect of time

Figure 10 shows the performance of the indexes as a function of time. Overall, the performance of the S^{e}TPR*-tree is the least affected by time and also consistently offers the best update and query performance. This is because shared query execution is less sensitive to changes in workload. The RUM*-tree has better update performance compared to the B^x-tree and the TPR*-tree. The main reason is that the RUM*-tree delays deletions and performs them in a batch which is more efficient. However, due to delayed deletions, the MBRs are tightened less frequently which explains its poor query performance. The B^x-tree shows periodical patterns for both update and query performance due to its use of dual-tree structures. The B^x-tree has the most query



(a) Query I/O

(b) Query execution time

(c) Update I/O

(d) Update execution time

FIGURE 10. Effect of time

execution time even though it produces less query I/O. This is because reduced I/O costs of the B^x-tree are eroded by its high CPU time for transforming a two-dimensional query range to an enlarged one-dimensional interval.

6.5. Effect of maximum object speed

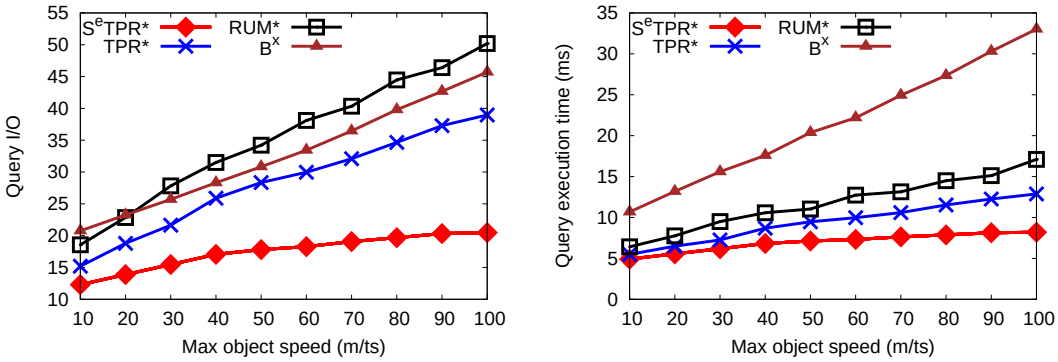
In this experiment, we study the effect of maximum object speed on the update and query performance among all the indexes by varying the maximum object speed from 10m/ts to 100m/ts. The results in Figure 11 show that the query performance of the S^{e}TPR*-tree is much less sensitive to object speed compared to the other indexes tested. The reason for this is that the S^{e}TPR*-tree's use of shared query execution is much less sensitive to MBR overlap (as explained in Section 3.2.) MBR overlap increases as speed increases. When the maximum object speed reaches 100m/ts, the S^{e}TPR*-tree outperforms the TPR*-tree by a factor of 1.9 for query I/O and a factor of 1.6 for query execution time.

The S^{e}TPR*-tree exhibits the lowest update costs compared to all other indexes. The S^{e}TPR*-tree has 2.8 times less update I/O and similarly for update execution time than the best of the three indexes (RUM*-tree) for updates. The S^{e}TPR*-tree and the RUM*-tree are barely affected by increasing object speed. The update cost of the TPR*-tree increases rapidly with increasing speed since it needs to do a lot of traversals of the R*-tree during update. Traversal costs increase with increased speed since higher speed results in more MBR overlap. The B^x-tree's update performance is not affected by increasing object speed as it does not take into account the object velocities.

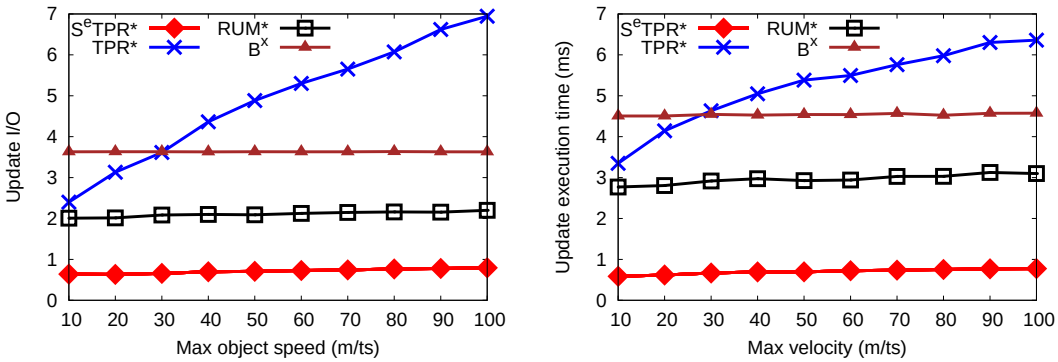
6.6. Effect of update frequency

In this experiment, the update frequency is varied from 1 to 10. This means that within a time interval of 120 time stamps, each object issues at least 1 to 10 updates at random time instances. Figure 12 shows that all the indexes have better update and query performance when updates are more frequent. This is because all indexes re-optimize the tree during updates. The S^{e}TPR*-tree has the best query performance and is by far the least sensitive to update frequency. The query performance of the RUM*-tree, the TPR*-tree and the B^x-tree improves rapidly with more frequent updates. This is an important observation since it means that the S^{e}TPR*-tree is much more robust to the frequency of updates compared to the other algorithms. We observed that when the updates frequency is 1, the S^{e}TPR*-tree outperforms the TPR*-tree by a factor of 1.9 for query I/O and 1.6 for query execution time.

For update performance, the S^{e}TPR*-tree also has the smallest update costs and is again hardly affected by update frequency. The results show that the S^{e}TPR*-tree outperforms the RUM*-tree (its nearest competitor) by a factor of 2.8 for update I/O and 1.9 for update execution

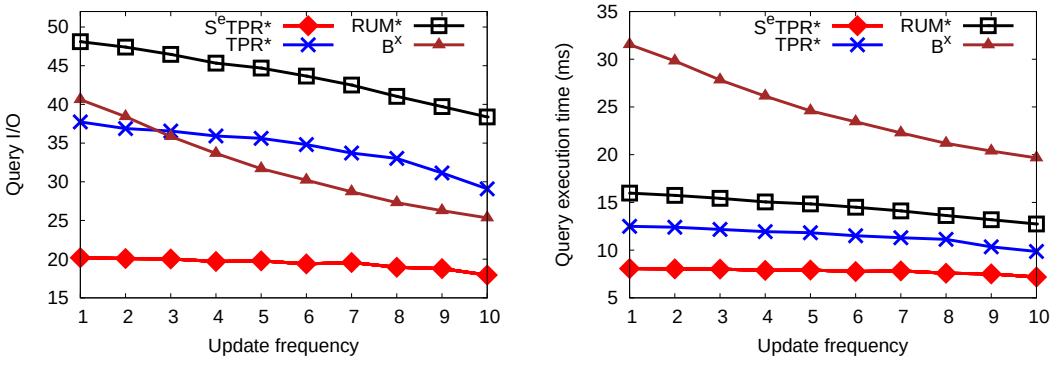


(a) Query I/O (b) Query execution time

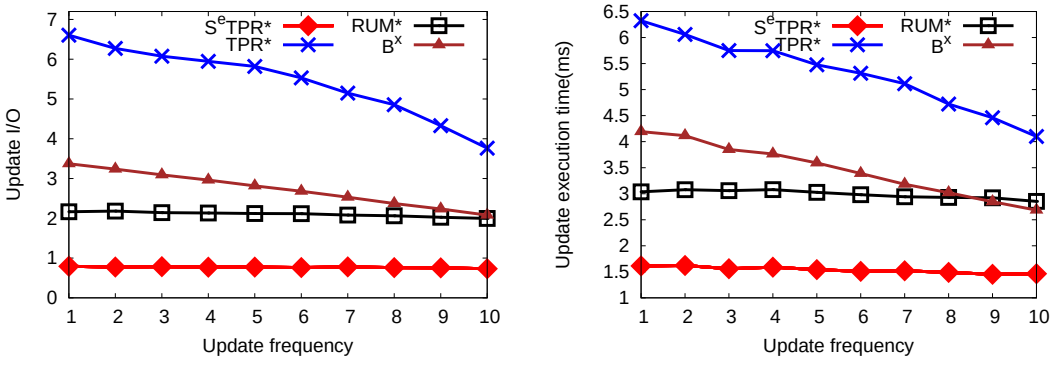


(c) Update I/O (d) Update execution time

FIGURE 11. Effect of maximum object speed



(a) Query I/O (b) Query execution time



(c) Update I/O (d) Update execution time

FIGURE 12. Effect of update frequency

time. The B^x-tree has worse update performance but is also largely unaffected by the update frequency. The TPR*-tree and the RUM*-tree have the most improvement in update costs with more frequent updates. These improvements are mainly caused by more frequent MBR tightening, and thus, the overlap among MBRs are less, which leads to better update performance.

6.7. Effect of RAM buffer size

In this experiment, we varied the number of RAM buffer pages from 4 to 256. The results in Figure 13 show that the RAM buffer size has a significant effect on query performance for all the indexes except for the S^{e}TPR*-tree. This is because the S^{e}TPR*-tree index uses only one RAM buffer page for querying. Therefore, its query performance is not sensitive to RAM buffer size. This result is very significant since it means that the query performance of the S^{e}TPR*-tree is much more robust to different RAM buffer sizes compared to the other indexes. The results show that the S^{e}TPR*-tree outperforms its nearest competitor, the TPR*-tree, by up to a factor of 2 for query I/O and a factor of 1.6 for total execution time.

For update performance, overall the S^{e}TPR*-tree significantly outperforms the other indexes. This is because the shared deletion execution and proximity ordered insertion use the RAM buffer much more efficiently. However, when the RAM buffer size is only 4 pages, the S^{e}TPR*-tree performs slightly worse than the RUM*-tree and B^x-tree. This is because the proximity ordered insertion and shared deletion execution need slightly larger RAM buffers to work at their maximum efficiency. The results show that the S^{e}TPR*-tree outperforms its nearest competitor, the RUM*-tree, by up to a factor of 11.5 for update I/O and a factor of 2.3 for update execution time.

6.8. Effect of range query size

In this experiment, we vary the size of the square window query from 2,000 x 2,000m^2 to 20,000 x 20,000m^2. Figure 14 shows that the query performance of all the indexes degrades rapidly with an enlarged query window, except for the S^{e}TPR*-tree which is close to constant (up to 25 I/Os). The reason is that a larger query window for the B^x-tree, RUM*-tree and TPR*-tree results in more node accesses. The reason the S^{e}TPR*-tree is much less sensitive to query window size is explained in Section 3, this being the cost of shared query execution equals the number of distinct node assesses. In all experiments, the disk-based TPR*-tree of our S^{e}TPR*-tree index contains less than 750 distinct nodes, therefore for the default 25 queries executed in a batch, the average I/O per query cannot be greater than 30. The results show that the S^{e}TPR*-tree outperforms the best tested index optimized for query (TPR*-tree) by up to a factor 3 for I/O and 2.5 for execution time.

6.9. Performance of k nearest neighbour queries

In this experiment, we examine the performance of the k nearest neighbour query instead of the range query. We vary the number of nearest neighbours from 100 to 1000. All other parameters used are the same as for the range query, namely the default parameters listed in Table 1. Figure 15 shows that for all indexes, I/O cost and query execution time increases with increasing k. The reason is a larger k leads to a larger effective search area.

The B^x-tree has the worst kNN query performance in both query I/O and execution time. The reason is that the B^x-tree processes a kNN query by using progressively larger range queries. This tends to result in it searching over a larger area than necessary.

The results show that the S^{e}TPR*-tree outperforms all other indexes in both query I/O and execution time. This is because the shared execution results in higher temporal locality of the reference stream. However, we note that the margin by which S^{e}TPR*-tree outperforms the TPR*-tree and RUM*-tree is lower than for the range query. The reason for this is the high density of objects in the query search space results in the query being effectively small in size (i.e. small search area).

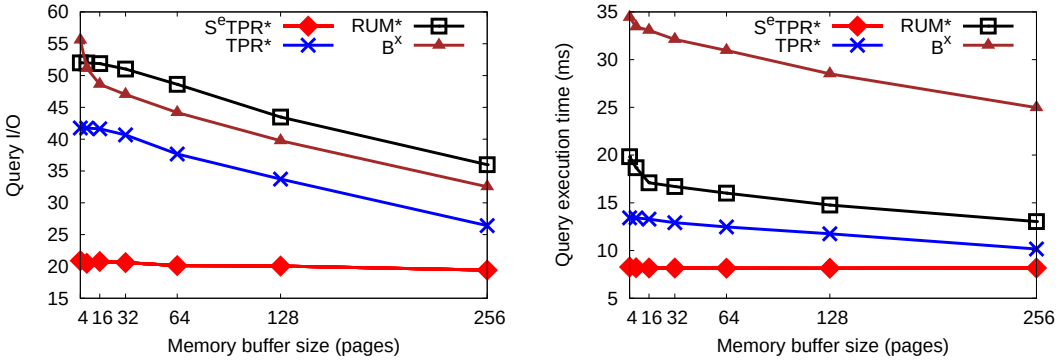
6.10. Throughput and response time

In these experiments, we measure both the throughput and response time for the range query. The throughput is defined as the average number of queries that can be processed within a second, whereas the response time is defined as the time it takes for a query to be answered from the time it is issued. As noted in the introduction, there is a tradeoff between throughput and response time. In these experiments, we will explore this tradeoff within the context of our algorithms.

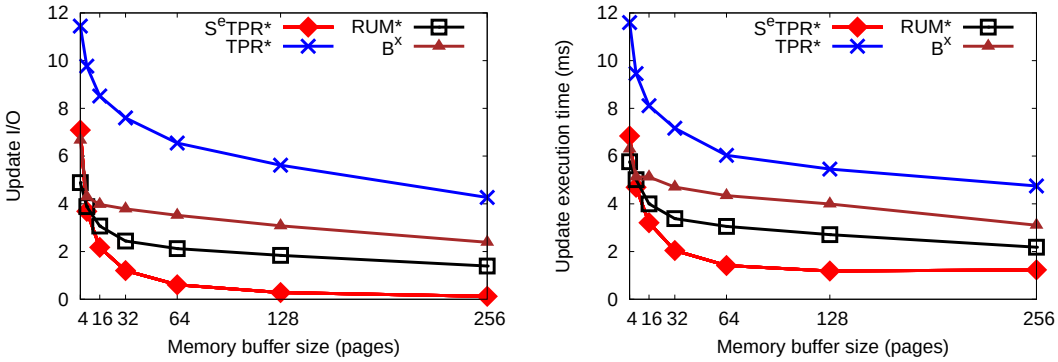
The results in Figure 16(a) show the effect on throughput as batch size increases. The results show that as the batch size increases, throughout of the S^{e}TPR*-tree increases significantly. The reason is that with bigger batch sizes, the average query execution time to process one query is significantly reduced. We observed that when the S^{e}TPR*-tree processes a batch of 100 queries, it has 3.7 times higher throughout compared to the best other index (TPR*-tree).

Figure 16(b) shows the response time results. The response time results shown are averages from performing 100 queries. This graph shows the effect of varying both the query arrival interval and batch size. The query arrival interval is varied on the x-axis and the results of different batch sizes are shown for the S^{e}TPR*-tree. The query arrival interval is important in determining the response time since it determines how much free time is available between query arrivals for the server to process the queries. A smaller arrival interval means there will be a high response time since there is less time between queries for query processing.

In this experiment, we vary query arrival interval from 0, 2, 4, ..., 20, where a value of 0 means that all 100 queries arrive at the same time, whereas a value of 2 means that one query arrives every 2 ms. A newly arriving query needs

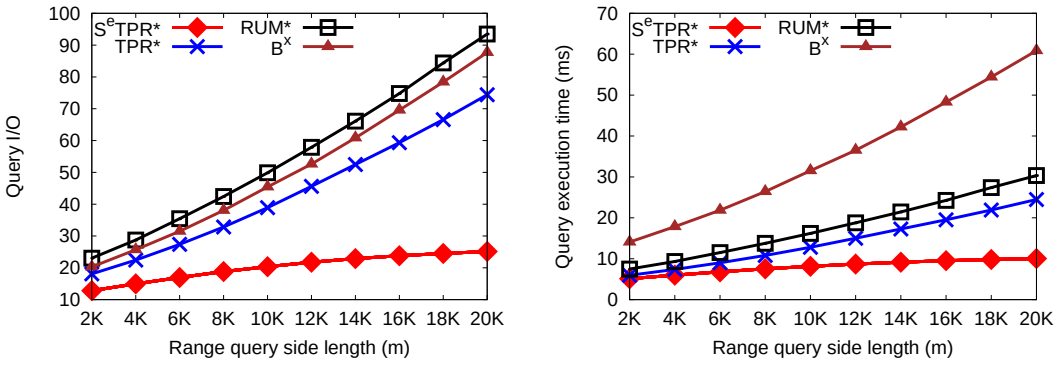


(a) Query I/O (b) Query execution time



(c) Update I/O (d) Update execution time

FIGURE 13. Effect of RAM buffer size



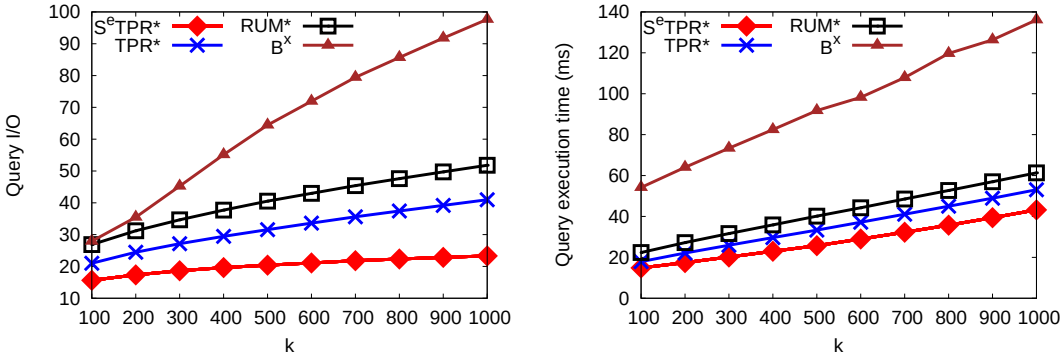
(a) Query I/O (b) Query execution time

FIGURE 14. Effect of range query size

to be placed in a queue if the previous query has not been completely processed.

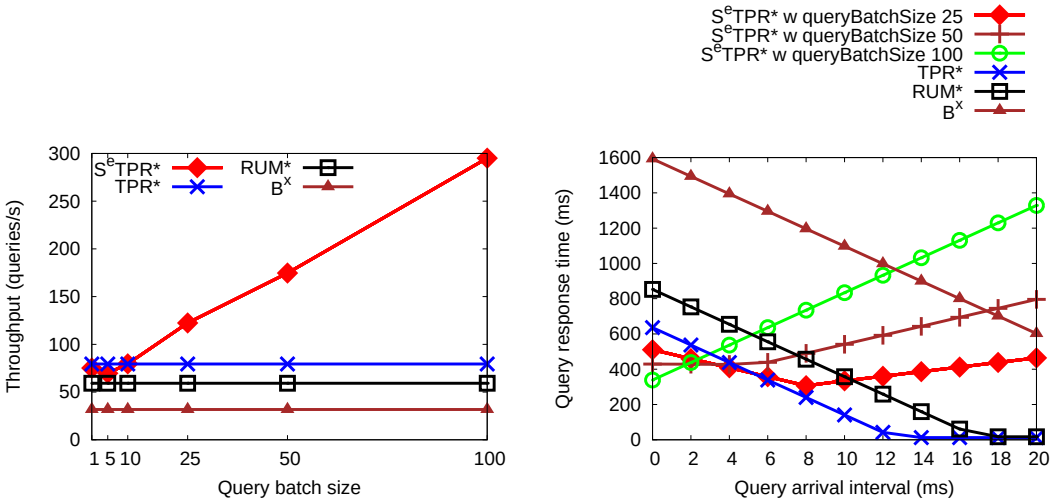
The results in Figure 16(b) show that the response time of the S^{e}TPR*-tree, in general, increases with increasing query arrival time. The reason for this is that in these experiments we force S^{e}TPR*-tree to wait for a batch of queries of a specified size to arrive first before processing the queries. Hence, there is a longer wait if the arrival interval is large. We note this is effectively artificially handicapping our system, but in real life, we would process whatever currently resides in the query queue straight away. So our system would work better in a situation where queries are arriving too fast for the system to handle.

The results show that the S^{e}TPR*-tree shows lower response time for smaller batch sizes when the query arrival interval is greater than zero. This can be explained by the fact that it needs to wait a shorter time period for smaller batch sizes before it can start processing. When the query arrival interval is zero, larger batch sizes result in smaller response time. This is because when all the queries arrive at once, processing the queries in smaller batches results in the later processed queries to have a very high response time, thus reducing the average response time.



(a) Query I/O (b) Query execution time

FIGURE 15. Effect of number of nearest neighbours



(a) Effect of query batch size on throughput (b) Effect of query arrival interval on response time

FIGURE 16. Throughput and response time

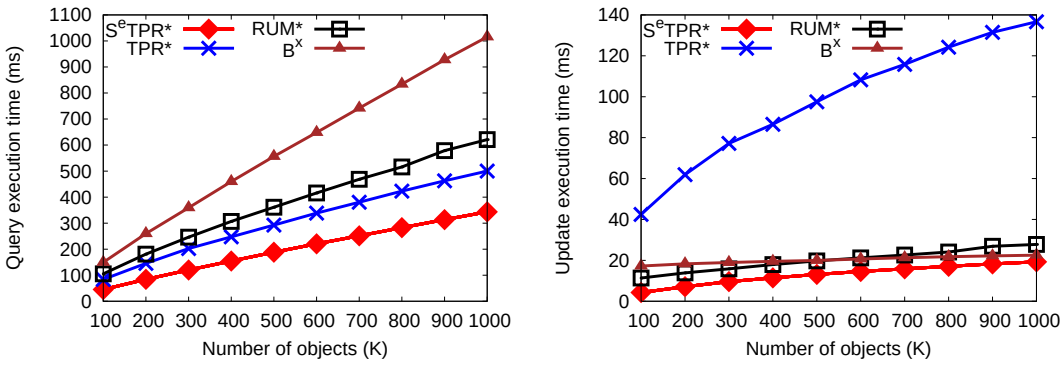
6.11. Vary data size using traditional hard disk

In this experiment, we examine the update and query execution time performance of the indexes under a traditional magnetic hard disk instead of SDDs. We use the same data sets and default settings as in Section 6.3 (Effect of data size). The hard disk used for this experiment is the Seagate Barracuda ST31000528AS 1TB 7200 RPM. The results are shown in Figure 17. The results show the same performance trends among the different algorithms as for the SSD results when the data set size is varied (see Section 6.3). However, the query and update execution time of all indexes using the traditional hard disk is up to the factor of 8 worse than the performance when using the SSD. This is because SSDs are much faster than magnetic hard disks in terms of both random access and seek time. The results show the S^{e}TPR*-tree still consistently outperforms all other indexes in both query and update performance.

7. CONCLUSION

This paper is the first to comprehensively study the impact of improving temporal locality for the performance of spatiotemporal indexes. Specifically, we proposed the S^{e}TPR*-tree which uses the following three techniques to improve the temporal locality of the page reference stream: shared query execution; shared deletion execution; and proximity ordered insertion. Integrating these three techniques with dynamic buffer allocation has resulted in a very effective use of the limited-sized RAM buffer. The consequence is significant I/O performance improvement over existing algorithms with small RAM buffer sizes.

In our experiments, we showed that the S^{e}TPR*-tree only needs a query batch size of 5 to outperform the best tested index (TPR*-tree) for query I/O performance. This means we only need relatively small query batch sizes to make shared query execution an effective technique for reducing query I/O. When we tested our shared deletion execution and proximity ordered insertion in isolation, the results showed shared deletion execution was responsible for the majority of the improvement in update I/O performance. When we



(a) Query execution time (b) Update execution time

FIGURE 17. Vary data size using a traditional hard disk

varied the data size, we found the S^{e}TPR*-tree consistently outperformed the other indexes for the entire range of data sizes tested. The results for varying maximum object speed showed that the S^{e}TPR*-tree was a lot less sensitive to MBR overlap compared to other indexes. Results showed that unlike the other indexes, the S^{e}TPR*-tree did not need frequent updates to improve query performance.

In future work, we would like to explore techniques for dynamically adjusting the index to favor processing individual queries versus batches of queries based on query workload characteristics. We would also like to explore our temporal locality enhancing techniques on the B$^+$-tree based spatiotemporal indexes.

ACKNOWLEDGMENT

We would like to thank Chen et al. [27] for providing us with the source code [28] of their benchmark, which we used in our experimental study. This work is supported under the Australian Research Council's Discovery funding scheme (project number DP0985451).

REFERENCES

[1] Saltenis, S., Jensen, C., Leutenegger, S., and Lopez, M. (2000) Indexing the positions of continuously moving objects. *Proceedings of ACM SIGMOD '00, Dallas, Texas, US*, 16-18 May, pp. 331–342. ACM, New York, NY, USA.

[2] Tao, Y., Papadias, D., and Sun, J. (2003) The TPR*-tree: an optimized spatio-temporal access method for predictive queries. *Proceedings of VLDB '03, Berlin, German*, 9-12 September, pp. 790–810. Morgan Kaufmann, San Francisco, CA, USA.

[3] Yiu, M., Tao, Y., and Mamoulis, N. (2008) The B^{dual}-tree: indexing moving objects by space filling curves in the dual space. *VLDB Journal*, **17**, 379–400.

[4] Jensen, C. S., Lin, D., and Ooi, B. C. (2004) Query and update efficient B^+-tree based indexing of moving objects. *Proceedings of VLDB '04, Toronto, Canada*, 31 August - 3 September, pp. 768–779. Morgan Kaufmann, San Francisco, CA, USA.

[5] Chen, S., Ooi, B. C., Tan, K.-L., and Nascimento, M. A. (2008) ST^2B-tree: a self-tunable spatio-temporal B^+-tree index for moving objects. *Proceedings of the ACM SIGMOD '08, Vancouver, Canada*, 10-12 June, pp. 29–42. ACM, New York, NY, USA.

[6] Zhang, R., Lin, D., Ramamohanarao, K., and Bertino, E. (2008) Continuous intersection joins over moving objects. *Proceedings of ICDE '08, Cancun, Mexico*, 7-12 April, pp. 863–872. IEEE Computer Society, Washington DC, USA.

[7] Biveinis, L., Saltenis, S., and Jensen, C. S. (2007) Main-memory operation buffering for efficient R-tree update. *Proceedings of VLDB '07, Vienna, Austria*, 23-27 September, pp. 591–602. Morgan Kaufmann, San Francisco, CA, USA.

[8] Silva, Y. N., Xiong, X., and Aref, W. G. (2009) The RUM-tree: supporting frequent updates in R-trees using memos. *VLDB Journal*, **18**, 719–738.

[9] Lin, B. and Su, J. (2005) Handling frequent updates of moving objects. *Proceedings of CIKM '05, Bremen, Germany*, 31 October - 5 November, pp. 493–500. ACM, New York, NY, USA.

[10] Guttman, A. (1984) R-trees: a dynamic index structure for spatial searching. *Proceedings of ACM SIGMOD '84, Boston, Massachusetts, USA*, 18-21 June, pp. 47–57. ACM, New York, NY, USA.

[11] Kwon, D., Lee, S., and Lee, S. (2002) Indexing the current positions of moving objects using the lazy update R-tree. *Proceedings of the International Conference on Mobile Data Management MDM '02, Singapore*, 8-11 January, pp. 113–120. IEEE Computer Society, Los Alamitos, CA, USA.

[12] Lee, M.-L., Hsu, W., Jensen, C. S., and Teo, K. L. (2003) Supporting frequent updates in R-tree: A bottom-up approach. *Proceedings of VLDB '03, Berlin, Germany*, 9-12 September, pp. 608–619. Morgan Kaufmann, San Francisco, CA, USA.

[13] Berchtold, S., Böhm, C., and Kriegel, H.-P. (1998) Improving the query performance of high-dimensional index structures by bulk-load operations. *Proceedings of EDBT '98, Valencia, Spain*, 23-27 March, pp. 216–230. Springer-Verlag, London, UK.

[14] DeWitt, D. J., Kabra, N., Luo, J., Patel, J. M., and Yu, J.-B. (1994) Client-server paradise. *Proceedings of VLDB '94, Santiago de Chile, Chile*, 12-15 September, pp. 558–569. Morgan Kaufmann, San Francisco, CA, USA.

[15] Kamel, I. and Faloutsos, C. (1993) On packing R-trees. *Proceedings of CIKM '93, Washington DC, USA*, 1-5 November, pp. 490–499. ACM, New York, NY, USA.

[16] Leutenegger, S. T., Edgington, J. M., and Lopez, M. A. (1997) Str: A simple and efficient algorithm for R-tree packing. *Proceedings of ICDE '97, Birmingham, U.K*, 7-11 April, pp. 497–506. IEEE Computer Society, Washington DC, USA.

[17] Roussopoulos, N. and Leifker, D. (1985) Direct spatial search on pictorial databases using packed R-trees. *SIGMOD Record*, **14**, 17–31.

[18] Bercken, J. V. d., Seeger, B., and Widmayer, P. (1997) A generic approach to bulk loading multidimensional index structures. *Proceedings of VLDB '97, Athens, Greece*, 25-29 August, pp. 406–415. Morgan Kaufmann, San Francisco, CA, USA.

[19] García R, Y. J., López, M. A., and Leutenegger, S. T. (1998) A greedy algorithm for bulk loading R-trees. *Proceedings of the ACM international symposium on Advances in geographic information systems ACM-GIS '98, Washington DC, USA*, 6-7 November, pp. 163–164. ACM, New York, NY, USA.

[20] Lin, B. and Su, J. (2004) On bulk loading TPR-tree. *Proceedings of the International Conference on Mobile Data Management MDM '04, California, USA*, 19-22 January, pp. 114–124. IEEE Computer Society, Los Alamitos, CA, USA.

[21] Arge, L., Hinrichs, K. H., Vahrenhold, J., and Vitters, J. S. (2002) Efficient bulk operations on dynamic R-trees. *Algorithmica*, **33**, 104–128.

[22] Zhou, J. and Ross, K. A. (2003) Buffering accesses to memory-resident index structures. *Proceedings of VLDB '03, Berlin, Germany*, 9-12 September, pp. 405–416. Morgan Kaufmann, San Francisco, CA, USA.

[23] Park, H. K., Son, J. H., and Kim, M. H. (2005) Adaptive index management for future location-based queries. *Journal of Systems and Software*, **74**, 313–324.

[24] Park, H. K., Son, J. H., and Kim, M.-H. (2003) An efficient spatiotemporal indexing method for moving objects in mobile communication environments. *Proceedings of the International Conference on Mobile Data Management MDM '03, Melbourne, Australia*, 21-24 January, pp. 78–91. Springer-Verlag, London, UK.

[25] Tao, Y., Papadias, D., and Zhang, J. (2002) Cost models for overlapping and multiversion structures. *ACM Transactions on Database Systems*, **27**, 299–342.

[26] Varman, P. J. and Verma, R. M. (1997) An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, **9**, 391–409.

[27] Chen, S., Jensen, C. S., and Lin, D. (2008) A benchmark for evaluating moving object indexes. *Proceedings of VLDB '08, Auckland, New Zealand*, 24-30 August, pp. 1574–1585. Morgan Kaufmann, San Francisco, CA, USA.

[28] Chen, S., Jensen, C., and Lin, D. (2008). Source code. http://www.comp.nus.edu.sg/s̃pade/benchmark.

[29] Li, Y., He, B., Luo, Q., and Yi, K. (2009) Tree indexing on flash disks. *Proceedings of ICDE '09, Shanghai, China*, 29 March - 2 April, pp. 1303–1306. IEEE Computer Society, Washington DC, USA.