Trees or Grids? Indexing Moving Objects in Main Memory*

Darius Šidlauskas Simonas Šaltenis Christian W. Christiansen
Jan M. Johansen Donatas Šaulys
Department of Computer Science
Aalborg University, Denmark
{darius, simas, cwinther, jmjo01, donis}@cs.aau.dk

ABSTRACT

New application areas, such as location-based services, rely on the efficient management of large collections of mobile objects. Maintaining accurate, up-to-date positions of these objects results in massive update loads that must be supported by spatial indexing structures and main-memory indexes are usually necessary to provide high update performance. Traditionally, the R-tree and its variants were used for indexing spatial data, but most of the recent research assumes that a simple, uniform grid is the best choice for managing moving objects in main memory.

We perform an extensive experimental study to compare the two approaches on modern hardware. As the result of numerous design-and-experiment iterations, we propose the update- and query-efficient variants of the R-tree and the grid. The experiments with these indexes reveal a number of interesting insights. First, the coupling of a spatial index, grid or R-tree, with a secondary index on object IDs boosts the update performance significantly. Next, the R-tree, when combined with such a secondary index, can provide update performance competitive with the grid. Finally, the grid can compete with the R-tree in terms of the query performance and it is surprisingly robust to varying parameters of the workloads. In summary, the study shows that, in most cases, the choice of the index boils down to the issues such as the ease of implementation or the support for spatially extended objects.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*Access methods*; H.2.8 [**Database Management**]: Database applications—*Spatial databases and GIS*

Keywords

Grid, R-tree, indexing moving objects, main-memory indexes

1. INTRODUCTION

A large class of emerging applications rely on monitoring of continuously evolving phenomena using vast quantities of online sensors. For example, advanced Location-Based Services (LBS) or Intelligent Transport Systems (ITS) rely on monitoring of collections of moving objects. Increasingly widespread availability of mobile devices with integrated positioning technology, such as GPS, transforms these types of systems from research prototypes to widely-deployed, commercial applications. A recent example is Google Latitude[1], which gained over one million users just a week after it was launched.

The architecture of such systems includes a server and a collection of monitored objects which regularly send their updated positions to the server in order to keep the location information up to date. The LBSs query the server with spatial queries like ”which cars are currently located within a specified area?” To process such queries efficiently, the server has to maintain a spatial index that, in addition to speeding up the query processing, is also able to absorb all of the incoming updates.

The following scenario helps to understand the rate of the incoming location updates. Two million mobile users are moving in an urban area with an average speed of 30 km/h, (circa 8.3 m/s). If an accuracy of 50 meters is required, each moving object has to send a new update every 6 seconds on average, i.e., whenever it is about to move 50 meters away from the previously reported position. At the server, this results in one update every 3 microseconds. The rate increases further if more objects are tracked and/or higher accuracy is required. As hard drives, involving moving mechanical parts, become too slow for such applications, main-memory solutions have to be employed. This is enabled by dropping prices and increasing capacities of RAM chips.

Previous research on main-memory indexing focused mainly on query performance. For example, the versatile and query-efficient R-tree was made cache-conscious [16] to optimize the use of the fast CPU caches. More recently, a number of papers on continuous processing of spatial queries [6, 20, 27] suggest using simple, uniform grids to index spatial positions of moving objects. It is argued that such grids are much faster to update, as the classical R-tree is known for its poor update performance. On the other hand, one can expect that uniform grids may not be good to adapt to skew in the workloads. Unfortunately, to the best of our knowledge, there is no comprehensive experimental comparison of these main-memory structures.

*This work was supported by the Danish Research Council.

[1] http://www.google.com/latitude

ACM GIS '09, November 4-6, 2009. Seattle, WA, USA
Copyright 2009 ACM 978-1-60558-649-6/09/11 ...$10.00.

This paper systematically explores the performance of basic space- and data-partitioning indexes—the grid and the R-tree. The contribution of the paper is twofold. First, we identify the variants of these two indexes that are update- and query-efficient in main memory. Then, we report the results of extensive performance experiments comparing these indexes. In these experiments, run on modern hardware, we compare the indexes in terms of the performance of updates, range queries, and kNN queries.

To identify the efficient variants of the two indexes we explore several design options in a number of design-and-experiment iterations. Specifically, we show that by using the so-called bottom-up updates [18], i.e., coupling a primary index with a secondary index on object IDs (see Section 3.1), the grid and the R-tree achieve over fourfold and sixfold performance improvement, respectively, without significantly sacrificing the query performance. As a result, the improved versions of the indexes are able to process the incoming update in less than 1 microsecond on average (for the aforementioned scenario).

The rest of the paper is organized as follows. The classical grid and R-tree indexes are briefly presented in Section 2. Section 3 describes the proposed main-memory efficient variants of these indexes. Section 4 presents experimental setting and reports the results of the performance study. Section 5 gives an overview of the related work. We conclude in Section 6.

2. THE GRID AND THE R-TREE

As the main challenge in the considered applications is the processing of very high update rates, the maintenance cost of the employed spatial index has to be as low as possible. In addition, the choice of the index should enable exploiting the available body of research on the index-based spatial query processing. Therefore, we focus on well-known, simple, and CPU-light index structures: the grid and the R-tree. Note that the grid and the R-tree are representatives of the two broad classes of indexes—space and data partitioning indexes, respectively.

This section gives an overview of the basic versions of the two indexes. We focus mostly on how updates are performed. More details can be found in the provided references.

In this section and in the rest of the paper, we assume that each update is a five-tuple $(oid, oldx, oldy, x, y)$. It can be represented as a pair of the deletion of $(oldx, oldy)$ and the insertion of (x, y) for the object identified by oid. Note that the bottom-up update technique, as described in Section 3.1, reduces the update message to a three-tuple (oid, x, y).

2.1 The Grid

A fixed grid [3] is a simple space-partitioning index where a predefined monitored area is divided into rectangular cells. Objects with coordinates within the boundaries of a grid cell belong to that particular cell.

2.1.1 Structure

Since update performance is the main focus of this paper, we consider a constant, uniform grid with equally-sized cells. Such a grid requires minimal maintenance costs. Thus, no grid refinement or rebalancing is performed when objects move from one grid cell to another as it is done in adaptive grids, such as the grid file [21], or in hierarchical space partitioning methods, such as the quad tree [10].

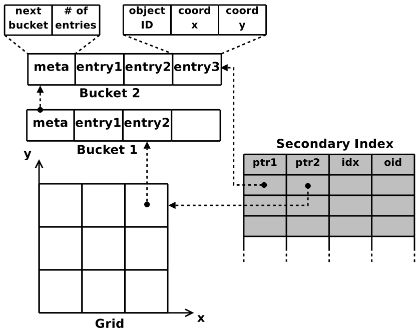

Figure 1: Grid index structure (the gray part is present only in the u-Grid)

Fig. 1 illustrates the overall design of the grid structure (ignore the gray component for now). The grid covers a predefined part of the coordinate space, *grid_area*. It is stored as a two-dimensional array, where each element of the array corresponds to a square spatial area with a side length of *gcs*. Each grid cell within the array stores a pointer to the linked list of buckets that contain the object data. The buckets have a fixed size, *bs*, and the grouping of objects into buckets follows no specific ordering. Thus, the grid is defined by the three parameters: *grid_area*, *gcs*, and *bs*.

Since the data to be processed during updating and querying is loaded in blocks (cache lines) to the CPU cache, when compared to a simple linked list of objects, large buckets increase the data access locality and enable the more effective prefetching of data by modern CPUs [9].

Each bucket has object data and meta data fields. The meta data field contains a pointer to the next bucket in the list and the current number of objects in the bucket.

2.1.2 Updates

All updates in the grid can be categorized as *local* or *non-local*. Any update involves determining the old cell of the object (using *oldx* and *oldy*) and the new cell (using x and y). If the old and the new cells are the same cell, the update is local and it involves scanning the buckets of the cell to locate the needed object and simply updating its x and y coordinates. The non-local update requires to delete the object from the old cell and insert into the new cell.

The insertion and deletion algorithms ensure that all except the first bucket of a grid cell are full. Thus, a new object is always inserted at the end of the first bucket. In case it is full, a new bucket is allocated and the necessary pointers are updated so that it becomes the first bucket. The deletion algorithm always moves the last object of the first bucket into the place of the object to be deleted. If the first bucket becomes empty, it is removed and the next bucket becomes the first or the grid cell becomes empty, storing a null pointer.

2.1.3 Queries

A range query is defined by a rectangle given by two corner points $(x_{q_{min}}, y_{q_{min}})$ and $(x_{q_{max}}, y_{q_{max}})$. The range query algorithm for the grid proceeds as follows. First, the cells covered by the range query are split into two groups—fully covered and partially covered. The objects from the fully covered cells are put into the result list by reading the corresponding buckets. The buckets from the partially covered cells are scanned and the objects are checked individually to determine whether they are within the range of the issued query.

A kNN query is defined by a query point, q, and the number of nearest neighbors required, k. A number of algorithms were proposed for performing kNN queries on grids [6, 27, 20]. All of them are based on a similar procedure. The underlying idea is that all cells are divided into different groups, such that cells within each group have similar minimum distances to the query point. Then, using a priority queue to store the cells, the cells are traversed, from the closest to the furthest, until k nearest objects are determined. We use an implementation of the algorithm recently described by Wu and Tan [25]. An array-based heap structure is used to implement the priority queue.

2.2 The R-tree

An R-tree-based indexing [12] and query processing has remained a focus of research in spatial databases for more than two decades now. The R-tree is known for its robustness to data skew, the support for a large number of different query algorithms, and its suitability for the indexing of spatially-extended objects (in addition to point objects). The main problem with the traditional R-tree is its poor update performance.

2.2.1 Structure

The R-tree is a balanced, data-partitioning tree index. It is a hierarchy of minimum bounding rectangles (MBRs). An MBR is the smallest rectangle that encloses a group of spatial objects. There are two types of nodes in the R-tree: internal nodes, which are nodes containing pointers to other nodes, and leaf nodes, which are nodes at the lowest level of the tree and contain moving objects. As illustrated in Fig. 2 (ignore the gray components for now), an internal node is composed of the node meta data (a number of entries and a leaf flag) followed by a number of child entries. Each internal node's entry has a pointer to a child and an MBR that encloses the objects within that child. Similarly, a leaf node is composed of the same meta data fields but followed by a number of leaf entries.

The main parameter defining the R-tree is the node size (ns). Note that we express it in cache lines. Another parameter is minimum children (mc), expressed as a fraction of the full node. In any node of the tree, its entries must occupy at least mc percent of ns. Otherwise, the node is considered underfull.

2.2.2 Updates

The classic R-tree processes each update as a combination of separate top-down deletion and insertion operations.

The deletion algorithm descends the tree from the root to the leaves, searching for $(oldx, oldy)$. This is done by recursively accessing the nodes with MBRs that contain $(oldx, oldy)$. Note that, due to possible overlap between the MBRs, more than one path down the tree may be visited. After the required leaf node is located, the appropriate entry is deleted. Finally, ancestor MBRs, which may become not minimum, are adjusted by traversing the tree up to the root. Furthermore, the nodes that are underfull have to be handled with an expensive reinsertion of their entries.

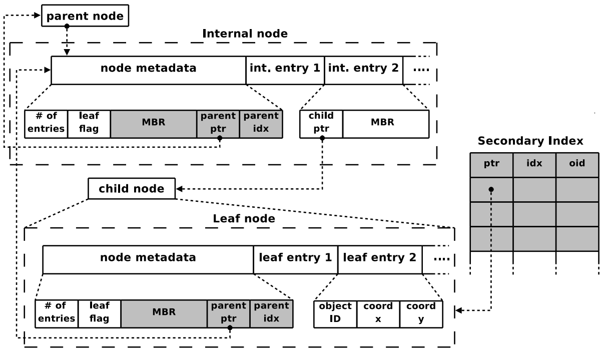

Figure 2: Structure of a conventional R-tree (the gray elements are present only in the u-R-tree)

The insertion algorithm begins by traversing the tree from the root to a leaf node as well. At each node, a heuristic function *choose_subtree* is called to choose the most suitable subtree to decent further. When a suitable leaf node for a new object is located, the object is inserted there. As ancestor MBRs may not be valid anymore, they are adjusted by traversing the tree back to the root. Insertion of a new leaf entry may also cause a node overflow which is handled by a node split algorithm. Because such a split produces an additional node and an additional parent entry in the parent node, the parent node may be split too and the split may propagate up the tree.

Summarizing, a single update operation results in four (possibly partial) tree traversals. This is the main reason for the inefficiency of the R-tree updates.

2.2.3 Queries

A range query in the R-tree is performed as a depth-first traversal from the root down to the leaves accessing the nodes with MBRs overlapping the query area. At the leaf nodes, objects satisfying the query are outputted.

A kNN query in the R-trees is processed as a best-first traversal [14]. Similarly to the kNN query processing in the grid index, it uses a priority queue storing the accessed MBRs organized on the minimum distance between an MBR and the query point. We use the same implementation of the priority queue both for the R-tree and for the grid.

3. UPDATE-EFFICIENT INDEXES

Having surveyed the classic grid and R-tree index structures, in this section, we present various design alternatives for improving the performance of these indexes in main memory.

3.1 Bottom-up Updates

As described above, in order to support fast range and kNN queries, the grid and the R-tree index the data on its spatial information, i.e., object coordinates make up the index key. This means that during the updates, the old object data must be located using spatial information $(oldx, oldy)$. In the grid, this means accessing the right grid cell and traversing a list of buckets associated with that cell. In the R-tree, this means doing a top-down traversal, possibly following several paths down the tree.

Furthermore, once the old position is deleted, the index is accessed again to insert the new position. The key property of the update workloads generated by most of the abovementioned applications is the *update locality*—the next update from an object is likely to be close to the previous one. Thus it is very probable that the new location have to be inserted in the same node or bucket as the old one.

To avoid the expensive top-down index searches and to leverage the update locality, we employ a *secondary index*, which uses *oid* as the index key and points to the location in a primary index associated with that key. In other words, it provides a direct access to the object's data in the primary index. This way, near constant *O(1)* updating time can be achieved. A hash-table is a suitable candidate for such secondary indexes, as it supports efficient insertions, deletions, and *oid* equality queries. This idea, called bottom-up updates was first proposed to speed up the updates of the disk-based indexes [17, 18]. Note that $(oldx, oldy)$ are no longer necessary in an update message.

3.1.1 Bottom-up Updates in the Grid

In Fig.1, the grid's secondary index is depicted in gray. An entry in the secondary index is composed of four fields. The pointer *ptr1* points to the bucket that contains the object, whereas *ptr2* points to the grid cell that contains that bucket. The field *idx* is an offset for object's position within the bucket.

When the new cell for the incoming update is determined, it is compared with the current cell referenced by the pointer *ptr2* to check whether the update moves the object to a different cell, i.e., whether the update is non-local. Then, *ptr1* and *idx* are used to compute the direct address of the entry to be updated. Note that no scanning of buckets is required, which is particularly desirable in grids with large cell and bucket sizes. If an update is non-local, the pointer *ptr2* is also used during the delete operation to move the last object from the first bucket of a grid cell into the bucket space of the deleted object (see Section 2.1.2). Note, that such moving of an object requires updating its *ptr1* and *idx* values in the secondary index, but that is a small cost to pay for the benefits of the secondary index.

3.1.2 Bottom-up Updates in the R-tree

To integrate the bottom-up updates in the R-tree, but not to complicate the already CPU-heavy algorithms too much, we use a simplified versions of the algorithms proposed by Lee et al [18].

First, the structure of the R-tree is augmented as shown in Fig. 2 (see the gray elements). That is, a backward pointer, *parent_ptr*, is added to each node such that the parent node can be identified without the prior top-down traversal. The added parent index, *parent_idx*, together with the parent pointer, provides a direct access to the corresponding parent's entry when MBR changes need to be propagated. In addition, a copy of the node's MBR in the meta data enables to know whether the MBR was invalidated without accessing the parent node, hence increasing memory-access locality and, thus, potentially decreasing CPU cache misses.

Similarly to the grid's updating, the exact object's place in the tree is determined using the pointer *ptr* and the offset *idx*. If the new object's coordinates are inside the leaf node's MBR and the old coordinates were not on the rectangle's boundary, then the update is carried out immediately by simply overwriting the outdated data. We call this kind of update a *pure-local update* (see Fig. 3). In case the old coordinates were on the MBR's boundary, the MBR must be shrunk and changes propagated up the tree (using parent pointers and *idx* indexes). Shrinking an MBR is an expensive operation since it involves scanning all the entries within the node to compute the new MBR. We call this kind of update a *shrinking-local update.*

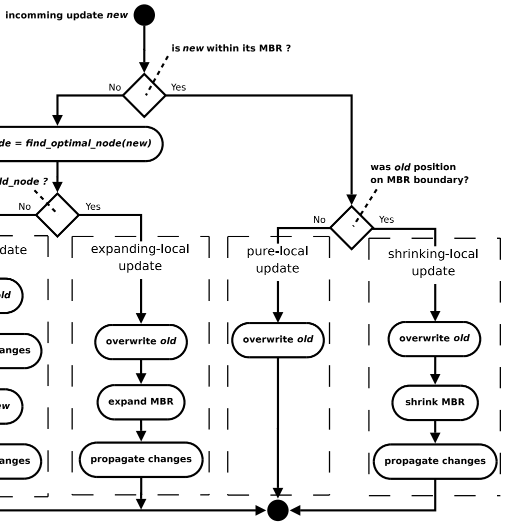

Figure 3: Bottom-up update in the u-R-tree

When the new object's position exceeds its current bounding rectangle, the tree is ascended looking for a less local solution (*find_optimal_node* procedure in Fig. 3). This bottom-up traversal stops when an MBR is found that covers the new position or the root node is reached. Then, starting from the node where the bottom-up traversal stopped, the tree is traversed down recursively using the *choose_subtree* algorithm, exactly as in the normal R-tree top-down insertion.

If the newly determined optimal leaf is the same as the leaf where the old position of the object is stored, the outdated data is overwritten and the bounding rectangle is expanded to include the new position. Finally, necessary MBR modifications are propagated up the tree as in the regular R-tree. We call this kind of update an *expanding-local update.* On the other hand, when the optimal node is different from the old one, a *non-local* update has to be carried out. This involves removing the object from its current leaf and inserting it into the optimal leaf, which, in turn, requires updating the secondary index and may cause node overflows and/or underflows.

Note that, while the parent pointer and the parent index speed up local updates, this is at the cost of the added maintenance of these fields during non-local updates. Specifically, when a node is split, half of the children get a new parent node (the split-off node). In case of a leaf node, the "children" are entries in the secondary index. Thus, the parent pointers and parent indexes of these children have to be updated. A similar procedure applies to merges of nodes (see Section 3.2). An alternative, middle-ground solution could be to have just a parent pointer without a parent index. Nevertheless, the experiments show that having *idx* pays off by eliminating the need to scan a node when looking for a particular entry. This is especially desirable since update processing favours the trees with large node sizes (see Section 4.4.1).

3.2 Node Splitting and Merging in the R-tree

The split algorithms of the advanced R-tree variants such as the R*-tree [2] are CPU heavy. On the other hand, the experiments show that the CPU-efficient Guttman's linear split algorithm tends to produce nodes with high spatial overlap. This is mainly due to a minimum node capacity constraint, which arbitrarily assigns a number of entries to the least populated node at the end of the split procedure.

The previous work [5] shows that linear splitting time and the query performance similar to that of an R*-tree [2] can be achieved using the k-means clustering algorithm. Our experiments confirm that and we use the k-means split algorithm for all our main-memory R-tree variants.

The R-tree algorithms handle an underfull node by deallocating it and reinserting its entries into the tree, but this is very expensive. To avoid this cost, underfull node is merged with a sibling node (with a possible split following the merge). The partner for merging is chosen by using the insertion's *choose_subtree* algorithm with an MBR of the underfull node and the parent node as the parameters.

3.3 Making the Secondary Index Primary

The idea of the bottom-up updates can be driven even further. Our experiments indicate that for our default workload (see Section 4.3), 90% of the updates are pure local updates in the R-tree.

To speed up the processing of these updates further, the object's data can be stored in the "secondary" index so that all the necessary information to process the local update is available after a single look up. The grid bucket or the R-tree leaf node then stores just *oid*s which are used to retrieve the data from the hash table when querying.

Note that in the case of the R-tree, in addition to the object's data, the entry of the secondary index has to store a copy of the MBR of the leaf node that contains this object. This is necessary, so that the type of the update can be determined without accessing the leaf's meta data in the R-tree. As our experiments show, the maintenance of this MBR overwhelms any performance gains (see Section 4.4.2).

3.4 Cache-conscious Techniques

The main idea in the design of cache-conscious indexes explored in recent studies is to pack more entries in an index node the size of which is close to the cache line size or a small factor thereof. For example, the so-called pointer elimination technique, applied in the CSB$^+$-tree [23], doubles the fanout, which leads to the reduction in the tree height. This, in turn, incurs less cache misses during the tree traversal. However, in the R-tree, MBRs and not child pointers occupy most of the index data and pointer elimination does not widen the tree significantly. Therefore, the authors of the cache-conscious R-tree (CR-tree) [16] employ the so-called relative representation and quantization techniques, which, in combination, reduce the MBR size to less than a fourth.

However, the more recent work [13] shows that the size of a node in the cache-conscious B$^+$-tree has to be much larger than the cache-line size. Our experiments, described in Section 4.4.1, confirm this. They show that main-memory R-trees benefit from even larger node sizes. This means that for realistic settings, the above-mentioned techniques for increasing the fanout do not usually lead to tree height reduction.

Table 1: Effects of the different cache-conscious techniques

Characteristic	R-tree	CSB$^+$-tree approach [23]	CR-tree approach [16]
Meta data	8B	12B	24B
Int. entry	20B	16B	8B
Leaf entry	12B	12B	12B
Int. fanout	25	31	61
Leaf fanout	42	41	40
#leaves req.	76,805	78,679	80,646
Int. height	$\lceil 3.97 \rceil$	$\lceil 3.7 \rceil$	$\lceil 3.03 \rceil$
Total height	**5**	**5**	**5**

Consider the following example. In our experiments with the kNN query processing, the R-tree exhibits the best performance when constructed with the node size of eight cache lines (512 bytes). The first column in Table 1 shows the observed tree characteristics after indexing 2 million moving objects. Note that the result is a five-level tree. The computation is performed assuming that, in the 32-bit system, the pointer size and each of the spatial coordinates occupy 4 bytes and the MBR is 16 bytes. This leads to 25 and 42 for internal and leaf fanouts, respectively. The average fullness of internal and leaf nodes was observed, in the experiments, to be 62 and 68 percent, respectively. Thus, we need 76,805 leaf nodes to store 2M objects ($\#objects/(leaf\_fanout \times 0.62)$) and, consequently, the tree height without the leaf level is 4 ($\lceil \log_{int.fanout \times 0.68} 76,805 \rceil = \lceil 3.97 \rceil$).

Based on these observations, the remaining two columns indicate that the increased fanout by the considered approaches can not widen the tree significantly. In CSB$^+$-tree approach, the child pointers in internal entry are eliminated (minus 4 bytes) and only the pointer to the first child is added in the meta data (plus 4 bytes). In the CR-tree approach, the reference MBR is added to the meta data (plus 16 bytes) and the regular MBR in the internal entry is replaced by a quantized relative MBR (minus 12 bytes)[2]. In spite of this, the total tree height remains five in both cases.

Since the ineffectiveness of these techniques can be already shown using the smallest optimal node size reported by any of the experiments (eight cache lines), we do not employ these techniques in our implementations, as our indexes are usually configured with much larger node sizes (see Section 4.4) where the effect is even more minimized. Moreover, note that these techniques would add an overhead in the insert/delete algorithms, resulting in higher update costs.

3.5 Summary

The following summarizes the index variants that are studied experimentally in the next section.

R-tree: the conventional R-tree index, with the improved handling of the underfull and overfull nodes (see Section 3.2).

u-R-tree: an update-efficient variant of the R-tree where a secondary index is employed to support the bottom-up

[2] Assuming we reduce the MBR size to a fourth, which is the best case [16].

updates (Fig. 2 with the gray elements).

u$^+$*-R-tree*: further u-R-tree optimization for update processing where all object data is stored in the secondary index rather than in the leaf nodes (see Section 3.3).

Grid: simple implementation of the fixed, uniform grid as described in Section 2.1.

u-Grid: an update efficient variant of the grid where a secondary index is employed to support the bottom-up updates (Fig. 1 with the gray elements).

u$^+$*-Grid*: a variant, similar to the u$^+$-R-tree, where all object data is stored in the secondary index rather than in the grid buckets (see Section 3.3).

4. EXPERIMENTAL STUDY

In this section we present the experimental study performed with the index variants designed in the previous section and summarize the main results.

4.1 Hardware and Software

All experiments were conducted on two machines, termed *system1* and *system2*, with Intel Core 2 Duo processors (2.20GHz and 2.60GHz) running under Linux 2.6.24 (32-bit and 64-bit variants) and each installed with 3.5 GB of RAM. The L1 cache is 32KB (16KB for instructions and 16KB for data), the unified L2 cache is 4MB and the cache line size is 64B for both machines. Note that although the CPU cache sizes are the same on both machines, due to 64-bit coordinates and pointers, the effective size of the cache in *system2* is reduced when measured in terms of the data elements.

The previously described spatial indexes and their variants were implemented in C++. In order to avoid unaligned accesses in main memory, the function `posix_memalign` (instead of the standard `malloc`) was used to allocate memory for data structures on a cache-line boundary. Additionally, a layout of each data structure was examined with the `pahole` program [7] to be sure all alignment holes were removed and better utilization of each cache line was achieved.

4.2 Measurement Tools and Methodology

In the cost drill-down experiments we avoid any approximations that simulation would impose by using hardware performance counters available on most modern microprocessors[3]. The architecture of the used processor provides five hardware counters for event measurement [1]. We employed the Performance API (PAPI [11]), a specification of a cross-platform interface to hardware performance counters, and the `perfctr` library to calculate the occurrence of various events. For instance, the performance metric *CPU time* was derived based on the count of CPU cycles and the processor's speed expressed in Hertz (*#cycles/cpu_speed*).

In order to show where each index operation spends most of the CPU cycles, the cycles are broken down into the following components: useful computation cycles (C_c), cycles stalled due to instruction and data misses at level 1 and level 2 caches (C_{L1i}, C_{L1d}, C_{L2i}, and C_{L2d}, respectively), cycles stalled due to instruction (C_{itlb}) and data (C_{dtlb}) misses in TLB (Translation Lookaside Buffer), cycles stalled due to branch misprediction (C_{br}), and cycles stalled due to other resource issues (C_o), e.g., instruction-length decoder stalls and dependency stalls. Thus, the following equation holds:

Table 2: Method of measuring each of the cost components (system1/system2)

Comp.	Method
C_c	(total cycles) - (cycles stalled on any resource)
C_{L1i}	excluded due to small code footprint
C_{L1d}	#misses * 2/2 cycles
C_{L2i}	excluded due to small code footprint
C_{L2d}	#misses * 64/39 cycles
C_{itlb}	excluded due to small code footprint
C_{dtlb}	#misses * 18/21 cycles
C_{br}	actual stalled cycles
C_o	(cycles stalled on any res.) - (all of the above)

$$C_{op} = C_c+C_{L1i}+C_{L1d}+C_{L2i}+C_{L2d}+C_{itlb}+C_{dtlb}+C_{br}+C_o$$

Table 2 shows a detailed list of the cost components and the way they were calculated. Since code footprint for each index operation is small and fits in the L1 instruction cache, the instruction-cache miss rate is very low (throughout the experiments, *#misses*/(*#hits* + *#misses*) never exceeded 0.1%). Thus, instruction misses at both cache levels and TLB were excluded. The latency of 2/2, 64/39, and 18/21 cycles for each of L1d, L2d, and data TLB misses on *system1/system2* were determined by using the Calibrator, a cache and TLB calibration tool [4].

In order to have the smallest possible overhead from any other operating system processes, the experiments were executed in run-level 1 mode of the operating system (single user, no GUI, no networking, no daemons, etc.) with the highest possible process priority. In addition, before taking any measurement, the main memory and caches were warmed up with multiple runs of the index operations as well as the counting instructions. All the experiments were run ten times and standard deviations and means were calculated for each index operation. Whenever the standard deviation divided by the mean was more than 1%, the experiment was rerun. Otherwise, the run with the lowest (least interrupted) counter values was recorded.

4.3 Workloads

The implemented indexes were exercised using a number of benchmark experiments, each defined by a set of workload parameters. A modified version of the COST benchmark [15] was used to generate the synthetic workloads so as to stress test the indexes under the controlled and varying conditions.

As a point of reference, we have identified a *default* workload with the settings that represent a scenario, which we consider realistic for the type of LBS application described in Section 1. The values for default workload parameters are shown in Table 3. Specifically, a given number of objects (*objects*) move in the two-dimensional monitored area (100×100 km^2). One of the maximum speeds $speed_i$ is randomly assigned to each moving object. The objects move between a given number of nodes (*hubs*) in the simulated road network. An object generates an update, whenever it moves a given accuracy *threshold* from its previously reported position. Four queries are issued every 2K updates. Equal amount of range and kNN queries are issued with a

[3] Although initially the cache simulator Cachegrind was employed to get a better insight of the cache behavior.

[4] http://www.cwi.nl/~manegold/Calibrator/

Table 3: Default workload parameters

Parameter	Value	Parameter	Value
objects	2M	$speed_i$	12, 25, 38, 50 m/s
threshold	100 m	*hubs*	500
selectivity	0.5%	*k*	100

Table 4: Optimal index parameters

Index	*gcs*	*bs*	Index	*ns*	*mc*
Grid	300 m	768 B	R-tree	14 cl	8 %
u-Grid	5500 m	128 B	u-R-tree	76 cl	10 %
u$^+$-Grid	5500 m	256 B	u$^+$-R-tree	37 cl	5 %
u-Grid(qe)	400 m	1 KB			

given queried fraction of the total data space (*selectivity*) for a range query and a given number of nearest neighbours required (*k*) for a kNN query.

4.4 Results

We proceed to describe the results of the performed experimental studies.

4.4.1 Determining Optimal Index Parameters

In the first set of experiments, using the default workload, the best parameter values are determined for the R-tree variants (*ns* and *mc*) and the grid variants (*gcs* and *bs*).

To determine the optimal values of *ns* and *mc*, the following tests were run. The index was exercised with *ns* values varying from 2 to 1200 cache lines and each of these node sizes was paired with a different value of *mc* (from 5 to 30%).

The results of such tests for the u-R-tree are depicted in Fig. 4. The left vertical axis represents the average CPU time performing a single operation (update, range query, or kNN query) while the horizontal axis shows different values for the node size in cache lines. For each different value of *mc* a separate line is used. The darker the line, the lower value it represents. Additionally, to have an idea of how the height of the constructed tree is changing, the blue line is drawn with the values on the right vertical axis.

It is not surprising that the bigger the node size is, the better update performance is achieved as huge nodes result in a larger fraction of local updates and fewer costly splitting operations. On the other hand, huge node sizes reduce the query performance for both types of queries. However, the graphs show that, in the range of up to 200 cache lines, it is possible to choose the index parameters so that performance is close to optimal for both updates and queries.

Fig. 4 shows that the *mc* parameter has a noticeable impact only in update processing. The trees configured with lower values tend to perform better, as the lower requirement for the minimum node capacity incurs fewer expensive merge operations, which is confirmed by the profiling data.

The similar experiments were run with the grid-based structures. Fig. 5 shows the results for the u-Grid. The grid cell size was varied from 50 to 5500 meters and run in combination with different capacities of the buckets (from 64 to 1024 bytes). Notably, the update-efficient values (for both *gcs* and *bs*) are completely different from the query-efficient values. The update performance benefits from more local updates caused by large cells (and smaller buckets), while querying benefits from larger buckets and cell sizes that match the average selectivity of the queries. Intuitively, the updates would be fastest with a single-cell grid, i.e., no grid at all.

Since update performance is much more critical in update-intensive applications, we configure the indexes with update-efficient parameters. Table 4 contains the chosen values for *system1*. For the 64-bit system *system2* (not shown), the graphs show similar patterns as those shown in Figures 4 and 5, but the optimal (chosen) values of the parameters tend to be slightly larger.

As mentioned above, the update- and query-efficient parameters for the u-Grid differ significantly, thus we also run experiments with the u-Grid configured to favour queries (u-Grid(qe)).

For the conventional grid and the R-tree, the observed optimal parameter values are relatively close to each other among all the three types of operations. That is, setting update-efficient parameters has almost no negative effects on the query performance. This is not surprising, since, without the benefit of the secondary index, both updates and queries traverse the same index structure. This explains why, in some experiments, the R-tree and the grid process queries faster than the other index variants with the update-efficient settings of the index parameters.

4.4.2 Comparison of the Indexes

In the following, the results of a number of experimental studies on varying workloads are presented. All figures show the average elapsed CPU time per index operation.

Study 1: Number of objects
Fig. 6 shows the results of the first experimental study under *system1*. Note the log scale on the vertical axis. As the number of indexed objects increases, the update performance degrades most rapidly for the conventional R-tree and the grid, while the bottom-up strategy significantly improves the update performance. This especially makes grid variants (u- and u$^+$-) a very scalable solution.

The processing cost of both query types is increasing for all indexes as the number of returned objects increases. As expected, the worst performance is shown by the u$^+$-variants. Another observation is that the optimization of the u$^+$-variants for local updates does not improve them in general by a significant margin. Quick profiling showed that the increased cost in processing non-local updates ties with the gains in processing local updates. As a result, no clear performance gain is achieved in updating while the querying is worsened significantly. This trend repeats in all of the conducted experiments.

Another interesting trend that also repeats in the following experiments is that the u-Grid, when configured with the query-efficient parameters (u-Grid(qe)), still performs updates slightly better and achieves very similar (range) and even better (kNN) query performance than the tree variants.

We could not find any qualitative differences between the results from both systems in all of the performed experiments. Thus, in the following, we show only the results under *system1*. In addition, since the conventional R-tree performs significantly worse in update processing, we exclude it from update graphs in the following figures. For the same reason, we exclude the u$^+$-Grid from both range and kNN query graphs.

Study 2: Position skew
In this study, the number of hubs in a simulated road net-

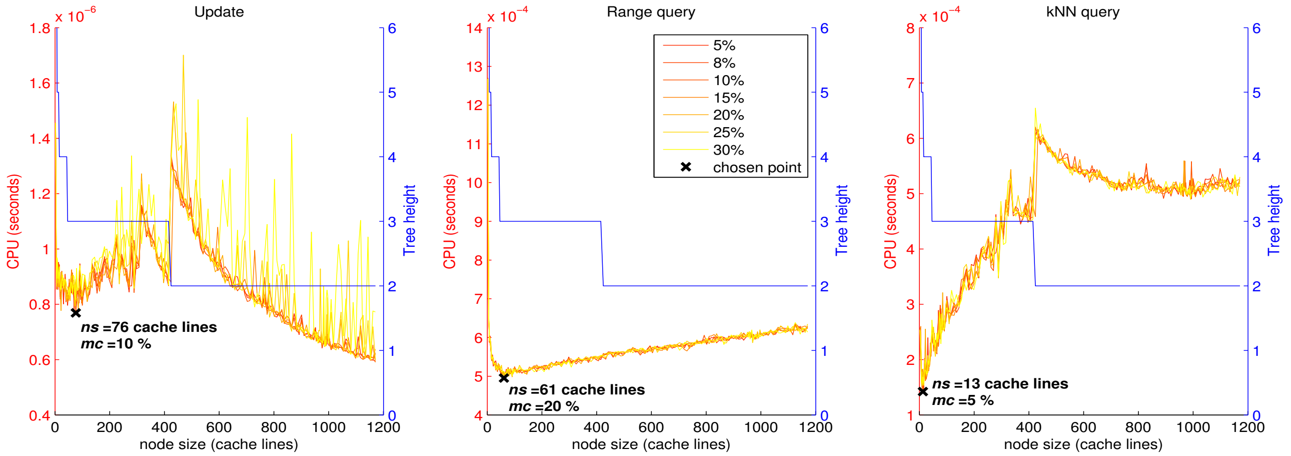

Figure 4: Determining optimal parameters for the u-R-tree

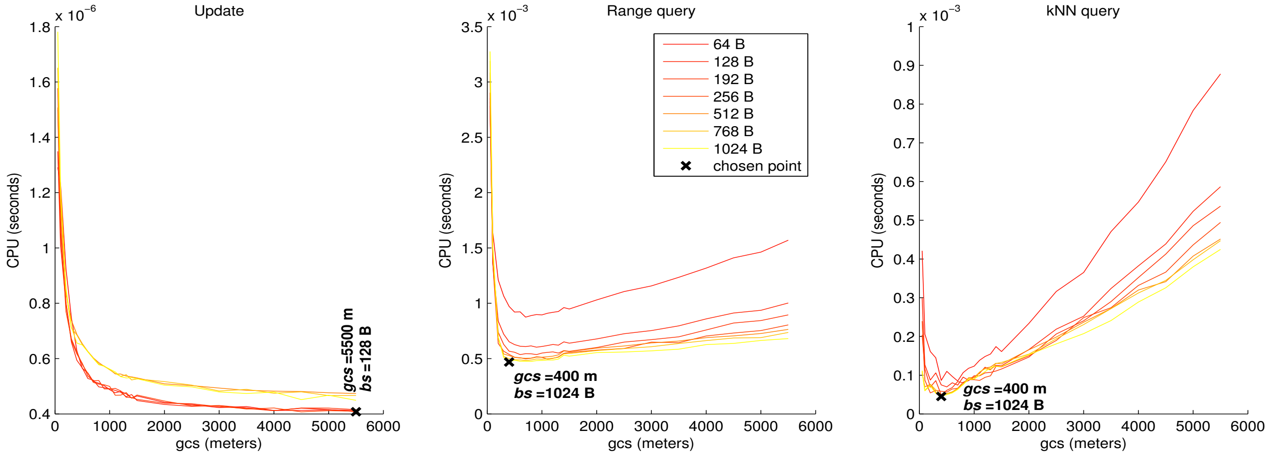

Figure 5: Determining optimal parameters for the u-Grid

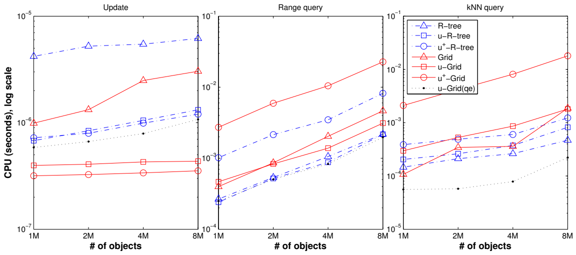

Figure 6: Increasing the number of objects

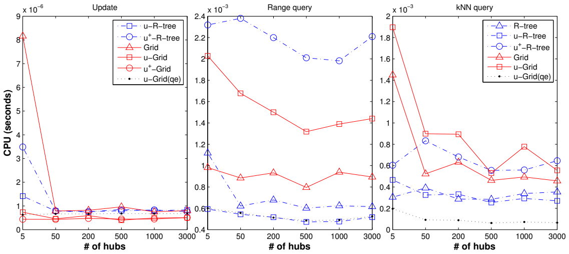

Figure 7: Increasing the position skewness

work is varied to change the distribution of spatial positions of objects from highly clustered (few hubs) to almost uniform (many hubs). Fig. 7 shows that in general all indexes tend to perform worse when objects are highly clustered (except for the improved grid variants). However, when the number of hubs exceeds 50, there is no significant difference between the indexes.

In terms of querying (both range and kNN), the better performance is exhibited by the R-tree variants, however, when the u-Grid is tuned to a given query workload, it outperforms the other indexes (see the graphs of the u-Grid(qe)).

Study 3: Position accuracy threshold

Varying the accuracy threshold values from 50 to 2000 meters results in a significant decrease of the fraction of (pure) local updates (e.g., from 94% to 17% in the u-R-tree). Fig. 8 shows that this mostly affects the R-tree-based indexes in update processing. When the accuracy threshold is more

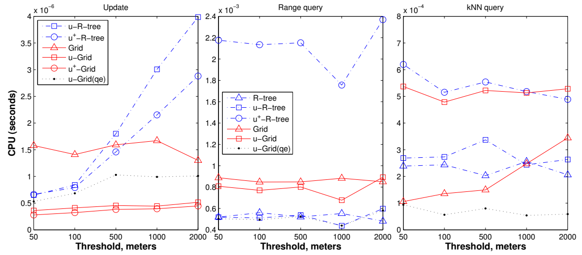

Figure 8: Increasing the number of non-local updates

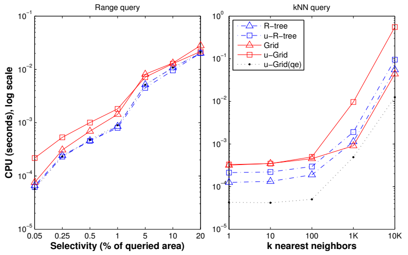

Figure 9: Varying the query parameters

than 500 meters (or less than 66% of the updates are local), any grid-based index outperforms the R-tree-based variant. The threshold does not seem to have any significant impact on querying.

Study 4: Query parameters

Fig. 9 depicts the behavior of the indexes when query parameters are varied. Unsurprisingly, the performance of the range and the kNN queries is degrading with the increasing selectivity and nearest neighbors required, as the number of objects to be returned is increasing. The steepest degradation is observed with very large *selectivity* (more than 1%) and k (more than 1K) parameters.

4.4.3 *Execution Time Breakdown*

In order to see where the different index variants spend their execution time, the cost of updates and queries was broken down into the cost components (see Section 4.2). Each bar in Figures 10 and 11 represents a contribution of the components as a number of cycles (bottom horizontal axis) and time (upper horizontal axis) spent during the execution of one operation.

Fig. 10 depicts such a breakdown for update processing. Three interesting observations can be made: (i) bottom-up updates reduces the computation cycles significantly (over twofold for the grid-based and almost tenfold for the tree-based indexes), (ii) the major bottleneck is the L2 data cache misses, and (iii) on average, the useful computation cycles take less than half, implying that most of the time the processor is stalled. A deeper analysis of the results indicates that the average number of L2d misses per update operation is very similar to the average number of L1d misses in the u- and u$^+$-variants of the indexes, implying that these misses involve the whole memory hierarchy and, thus, can not be avoided (compulsory misses). We hypothesize that most of the misses occur when performing a look-up in the hash-table.

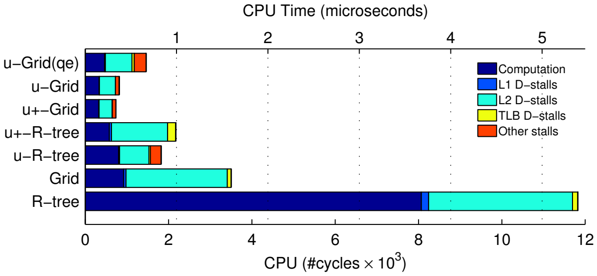

Figure 10: CPU cycles/time breakdown in updating

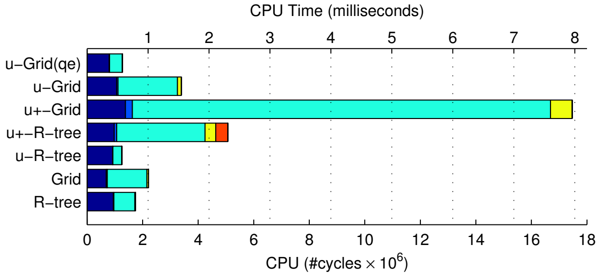

Figure 11: CPU cycles/time breakdown in range query processing

Fig. 11 shows the same breakdown for range queries. As expected, the huge performance difference between u- and u$^+$-variants in querying is purely due to increased L2 data cache misses, i.e., object's data storage in the secondary index does not add any computational overhead, but incurs waiting cycles for the data to be retrieved from the secondary index. At the same time, TLB data and other stalls become more apparent. In contrast to L2d misses in updating, for queries, the techniques such as prefetching might help to reduce the stalls. Similar cost distribution is observed in kNN queries (not shown).

Also, note that a proper configuration might reduce some of the L2d misses. For instance, consider how the contribution of the L2d misses in the u-Grid is reduced when it is configured with the query-efficient parameters (see u-Grid(qe) in Fig. 11), whereas this incurs unnecessary stalls in update processing (see u-Grid(qe) in Fig. 10).

5. RELATED WORK

A substantial body of research has accumulated concerning moving object indexing on disk, and recent surveys cover different aspects of this research [22, 19].

A popular technique to reduce the update rate and to provide predictions of near-future positions is using functions of time to approximate movement (e.g., the TPR-tree [24] indexes linear functions). However, the index operations become very CPU intensive. Thus, fewer but more expensive updates might not pay off. Therefore, in this paper we consider indexing simple (x,y)-coordinates of moving objects.

Since even a single disk I/O during update operation is a bottleneck in update-intensive applications, bulk-loading techniques look attractive. The idea is to accumulate the incoming updates and perform them in groups so that the cost is shared among several operations that work with the same page. Thus, various buffering strategies were applied for both disk-based grids [26] and R-trees [4]. In memory, such techniques were applied by Zhou and Ross [28]. Note that buffering strategies are orthogonal to the techniques used in this paper and can be applied to further enhance performance.

In our proposed main-memory index variants, we use the simplified version of the bottom-up update strategy which was successfully used in disk-based indexes [18]. As mentioned in Section 3.1, it is based on the observation that moving objects exhibit locality-preserving updates [17] which mostly lead to minimal changes in spatial index structure.

In recent works, the grid structure was a popular main-memory solution for continuous kNN query monitoring [6, 27, 20]. Similarly to the u$^+$-Grid, the work by Chon et al. [6] stores only object IDs in grid cells and uses a hash table for frequent random accesses. However, their main focus is continuous-query support. For tree-based indexes, in Section 3.4 we carefully considered the applicability of several existing cache-conscious techniques [16, 23].

Very recently, Dittrich et al. [8] proposed an interesting main-memory indexing method for moving objects called MOVIES. The approach is based on frequently building short-lived throwaway indexes where the query result staleness is traded for both update and query efficiency. We, in contrast, follow the traditional performance trade-off between updates and queries while delivering the most up-to-date positions to each query request.

6. CONCLUSIONS

Through a number of iterations of designing and experimenting, we identify update and query efficient variants of the well known grid and R-tree indexes in main memory. The extensive performance experiments with these indexes reveal a number of interesting insights.

First, the bottom-up updating, facilitated by the secondary hash index, significantly boosts the update performance. This makes the R-tree indexes competitive with the grid indexes for update processing. Next, the query performance of the uniform grid is surprisingly robust and competitive with the query performance of the R-tree. In summary, the two optimized versions of indexes are comparable in update and both range and kNN query performance.

Having observed that, we conclude that other factors need to be considered when choosing an index. For instance, the u-Grid is simpler than the u-R-tree. Thus, it is easier to implement. On the other hand, grids, in contrast to R-trees, require specifying a predefined spatial area. Also, R-trees, in contrast to grids, support spatially-extended objects which may be useful when modeling the inherently inaccurate positions of moving objects.

7. REFERENCES

[1] Intel 64 and IA-32 architectures software developer's manual. Intel Corporation, March 2009.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

[3] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[4] L. Biveinis, S. Šaltenis, and C. S. Jensen. Main-memory operation buffering for efficient r-tree update. In *VLDB*, pages 591–602, 2007.

[5] S. Brakatsoulas, D. Pfoser, and Y. Theodoridis. Revisiting r-tree construction principles. In *ADBIS*, pages 149–162, 2002.

[6] H. D. Chon, D. Agrawal, and A. El Abbadi. Range and knn query processing for moving objects in grid model. *Mob. Netw. Appl.*, 8(4):401–412, 2003.

[7] A. C. de Melo. The 7 dwarves: debugging information beyond gdb. In *Proc. of the Linux Symp.*, 2007.

[8] J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, pages 189–207, 2009.

[9] U. Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2008.

[10] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, March 1974.

[11] B. D. Garner, S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The Inter. Journal of High Perf. Comp. Appl.*, 14:189–204, 2000.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, 1984.

[13] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b+-trees. *SIGMETRICS Perform. Eval. Rev.*, 31(1):283–294, 2003.

[14] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[15] C. Jensen, D. Tiesyte, and N. Tradisauskas. The COST Benchmark–Comparison and evaluation of spatio-temporal indexes. In *Database Systems for Advanced Applications*, pages 125–140, 2006.

[16] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. *SIGMOD Rec.*, 30(2):139–150, 2001.

[17] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *MDM*, pages 113–120, 2002.

[18] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: a bottom-up approach. In *VLDB*, pages 608–619, 2003.

[19] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26:40–49, 2003.

[20] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.

[21] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[22] B. C. Ooi, K. L. Tan, and C. Yu. Frequent update and efficient retrieval: an oxymoron on moving object indexes? In *WISEW*, page 3, 2002.

[23] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, 2000.

[24] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[25] W. Wu and K.-L. Tan. isee: Efficient continuous k-nearest-neighbor monitoring over moving objects. In *SSDBM*, page 36, 2007.

[26] X. Xiong, M. F. Mokbel, and W. G. Aref. Lugrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, page 13, 2006.

[27] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.

[28] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, pages 405–416, 2003.