Department of Computer Science at Aalborg University

10th Semester, 12 June 2008

Update-Efficient Main-Memory Indexing of Moving Objects

Donatas Saulys (donis@cs.aau.dk), Jan Maffert Johansen (jmjo01@cs.aau.dk),
Christian Winther Christiansen (cwinther@cs.aau.dk)
Supervisor: Simonas Saltenis

Abstract The number of location-based services for monitoring and querying large numbers of moving objects is increasing rapidly. The services are facilitated by spatio-temporal databases that continuously receive updated position data from objects. This has inspired the development of indexing structures that perform well with dynamic spatio-temporal data, since traditional spatial indexing structures suffer from large update costs or poor query performance when used in spatio-temporal settings.

In recent years, memory prices have dropped, making main-memory databases a viable option. The transition from disk-based to memory-based databases means that existing knowledge on database index performance needs to be revised. Several index structure optimizations for memory implementation have therefore been proposed. Unfortunately, the results of individual proposals cannot easily be compared, as each proposal carries its own individual setting. This motivates the need for a performance benchmark of the most significant spatio-temporal indexing structures in a main-memory setting.

We modify an existing disk-based performance benchmark, enabling us to evaluate and compare main-memory spatio-temporal indexes. We propose main-memory variants of a Grid and an R-tree, along with a simple array structure, examine their performance in a variety of settings and compare the results.

We conclude that the Grid is generally the faster of the three indexes. Non-local updates in the R-tree are slower, because the tree structure has to be maintained. The Array performs the fastest updates and the slowest queries, although for some workloads, the combined performance is comparable. However, we believe that LBS end-users value fast queries, which is why the Array is not suitable for query efficient LBSs. Though the results show query times that favour the Grid, considering the higher versatility of the R-tree, we find that both indexes are highly suited for indexing the positions of dynamic spatio-temporal moving objects.

1 Introduction

Applications that monitor moving objects and provide Location Based Services (LBS) are becoming increasingly popular. The technology for such applications is based on the integration of mobile devices and global positioning units. The price and size of these devices has dropped dramatically in recent years, increasing availability and making the technology more widespread.

The architecture behind these applications comprises a server offering LBSs to users, and monitored objects which regularly send their updated positions to the server, in order to keep the location information up-to-date.

The monitored objects are typically cars, users of wireless devices, etc. which share the common characteristic that their geographical positions change over time. The LBSs query the server with spatio-temporal queries like: "which objects are currently located within a specified area?".

In order to maintain precise object location data and to prevent outdated answers to spatio-temporal queries, it is necessary to update this information very frequently. If the monitored area is a big city with hundreds of thousands of monitored objects, examining the location of every object for each query is not a feasible solution. Therefore, it is highly desirable to index the location of the objects, which speeds up the queries.

The challenge when using spatial indexing in this setting is the continuous change of object positions which necessitates continuous updating of the index. Using location update policies [16], [3], the number of updates required to maintain a certain accuracy can be reduced significantly. This way the load on the index structure is decreased. Advanced update policies can significantly reduce the number of updates by making predictions about future object movement. However, this reduction does not increase efficiency of the update processing within the indexes. We therefore use a simple update policy, and instead focus on examining how best to index moving objects.

Existing index structures were developed and optimized for disk-based environments. However, dropping prices of main-memory are making main-memory databases a viable alternative, facilitating indexes that meet the update and query performance requirements. In recent years, a number of main-memory indexing techniques have been proposed (e.g. the CR-tree [8], and the cache sensitive B^+-tree, CSB^+-tree [12]). Unfortunately, the empirical data provided as basis for conclusions about the performance of the proposed indexes is rarely exhaustive.

There are two main approaches for indexing moving objects. *Space* partitioning indexes and *data* partitioning indexes. The basic *space* partitioning index is a grid-based structure, where the monitored area is partitioned into cells that each contain objects, whose coordinates are within the range of that cell. The other option is to partition the *data* in some tree structure, e.g. the R-tree [5], which is easier, since no information about the underlying space is required. On the other hand, updates are more cumbersome, since the tree structure needs to be maintained.

Authors of spatio-temporal indexes typically offer empirical experiments, where the proposed index is compared to one or a few other competitive indexes. However, the experiments tend to be based on a setting which favours the proposed index, and focus on the most promising properties of that index, making the results ill-suited for broader conclusions.

As a result, it remains unclear whether the data or space partitioning approach is superior for indexing moving points in a main-memory setting. We therefore propose update-efficient main-memory versions of a Grid structure (representing the space partitioning approach) and an R-tree (representing the data partitioning approach), along with an implementation of a simple array structure, which serves as a base case for comparison.

We examine these indexing structures using a main-memory variant of the COST benchmark [7], in order to provide a performance comparison of the Grid and the R-tree. The COST benchmark offers a unified procedure covering a variety of spatio-temporal settings for moving objects tracking, and evaluates update and query CPU performance, which enables us to provide an independent evaluation of the proposed indexing structures.

The rest of the paper is organized as follows. Section 2 presents our problem setting. Section 3 reviews the related work. Section 4 offers an overview of the techniques and index structure types that we consider part of the solution space. Section 5 discusses design issues of our proposed indexing structures. Section 6 offers an overview of our implementation work. Sections 7 and 8 present the conducted experiments and their results. Section 9 concludes on the design and experiments.

2 Problem Setting

In this section, we define the representations of moving objects and the updates they generate. Since it is desirable to process as few updates as possible, update policies that reduce the amount of updates, while providing accuracy guarantees are discussed.

In order for an LBS to offer versatile services to users, it must support a variery of spatio-temporal queries, which are also defined in this section. Finally, the challenges of implementing indexes in main-memory instead of disk are discussed.

2.1 Moving Objects

We model moving objects in 2-dimensional space. The position coordinates of individual objects are aquired using GPS or some other positioning system. These systems typically provide an accuracy of approx. 5-10 meters. Since a more fine-grained positioning is rarely needed in this type of applications, we assume the received position information to be accurate.

The monitored area is mapped to a two-dimensional coordinate system with dimensions x and y, where object positions are represented as points with (x,y)-coordinates.

Moving objects change their position over time. The object movement is defined as a vector in (x,y)-space, which represents the object velocity. This velocity vector is required for performing predictive queries (described in Section 2.3).

2.1.1 Updates

In order for the LBS to stay up-to-date on current object locations, it is necessary to send updates to the server. Updates are of the following form:

oid: Unique object ID.

coord: New (x,y)-coordinates of the object.

v: New (v_x,v_y)-velocity vectors of the object.

Upon receiving an update, the server updates the existing object position information.

2.2 Update Policies

There are several strategies for updating the server, which we will examine below.

2.2.1 Time-based update policy

One update policy is that each object sends an update every x time units, e.g. 5 seconds. It is assumed that no objects will move enough to make the data on the server outdated or unacceptably inaccurate within that period.

This policy has some undesirable side-effects:

Server costs If we monitor 100,000 objects at 5 second intervals, they generate 20,000 updates per second. Current DBMSs are not designed to handle such large volumes of updates - particularly not if objects are spatially indexed.

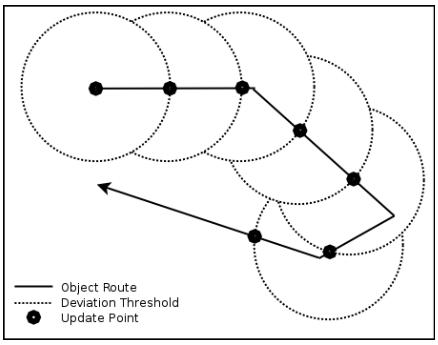


Figure 1: Point-based update policy. When the object is more than some Δ from its last stored location it triggers an update.

Client costs Communication on wireless networks induces a significant computation overhead on mobile devices. Since devices are typically battery-powered, this causes shorter operation time.

Network costs The large volume of updates may challenge the performance of wireless networks. Furthermore, transmitting data via mobile operator networks remains rather expensive, although cheap flatrate 3G subsciption solutions are emerging.

The number of updates *can* be reduced significantly. For example, when cars move, their position needs to be updated, but most of the time they remain stationary, and it is not necessary to send any updates during these periods.

Another issue is that time is not the most suitable measure in this setting. Essentially, if the x time units between updates is too low, too many updates are generated, leading to the mentioned side-effects. If x is too high, accuracy is insufficient. Besides, even the smallest time interval between updates does not provide any guarantee as to how far the object has moved since the last update. The deviation can be approximated if a global maximum speed is included in the calculation since it defines the maximum possible movement from the last stored location, but this requires knowledge about the maximum speeds of the monitored objects.

Time-based update policies do not allow for reductions in updates without loss of accuracy, because they are triggered by changes in time, which is constant.

If instead updates are triggered by change in space, then the number of updates can be reduced while still maintainging some accuracy guarantee.

2.2.2 Point-based update policy

The point-based policy, introduced by Civilis et al. [3], uses the fact that objects are location-aware, so that updates are only sent when the object moves beyond a given position threshold. In the point-based update policy, an objects position is represented as the most recently stored location (x_{stored}, y_{stored}) and the distance from this location is calculated based on (x_{upd}, y_{upd}). An update is triggered if the object moves more than some distance threshold:

$$\Delta = \sqrt{(x_{upd} - x_{stored})^2 + (y_{upd} - y_{stored})^2}$$

from its last stored position, as illustrated in Figure 1.

This way, the number of updates can be significantly reduced. Updates are not sent when the object is stationary, and there is a guarantee that the object is located within a specific distance from its last stored position. The time of the last update is not stored, since at any given time, the object position will not have changed beyond the Δ threshold.

Other update policies like vector-based and segment-based were proposed by Civilis et al. [3]. They can further reduce the number of updates required to maintain a given Δ threshold guarantee, but since they introduce added complexity, and since update policies are not our main focus, we use only the point-based update policy in this paper.

2.3 Queries

There are various query types which an LBS can support. We support object queries, current-time and predictive range queries, along with kNN queries. In this section we define these queries and their semantics.

2.3.1 Object Query Semantics

Object queries take an object ID as a parameter to locate and return the position and velocity data of the corresponding object. The object ID is typically a unique identification number.

The object query is the first step of the update operation, where it is necessary to query the object in order to delete the old location information, before the updated information can be inserted. Since an architecture that scales to handle a large number of updates is required, performing fast object queries is essential.

Although the object queries are important, their semantics are simple, so they are not discussed further in this section.

2.3.2 Range Query Semantics

In spatio-temporal range queries we answer questions like: "Which objects are located within a specified area at a given time?". We have a a query area q, the simplest 2D shape being a rectangle which is defined by two points q_{min} and q_{max} and a query time t_q, where $t_{curr} \leq t_q$.

If the shape is a rectangle, cube or n-dimensional hypercube, it is defined by a minimum and maximum value in every dimension. This means that we can define a general formula for evaluating a query for a given dimension d.

All objects send updates and when a query is made, the object position d_q at the query time t_q is calculated as shown in Figure 2. The calculations use $t_{current}$ and the position and velocity of the object at the time of the query d_{stored} and $v_{d_{stored}}$. The query interval is defined as $[d_{q_{min}}, d_{q_{max}}]$.

There are current-time and predictive range queries. Considering the current-time case, with the time-based update policy, the last stored position could be used, but would be inaccurate. Projecting the last stored position into current-time using the velocity vectors would require a timestamp parameter to be added to each update, and would also require computing the projection, generating server overhead.

However, when using the point-based update policy, updates occur only when an object moves beyond the distance threshold Δ of its last update position. The update policy provides the guarantee that until an object updates, it remains located within the threshold of the position stored from the last update, that is (in the d'th dimension):

$$d_q \ \approx_\Delta \ d_{stored}$$

Hence, the last stored object position is also used to indicate the current object position, since that is assumed to be within the threshold.

For predictive range queries, it is necessary to project the current position into the future. The temporal offset used to calculate the projected position, t_{offset}, is the difference between the current time, $t_{current}$, and the query time, t_q, i.e. :

$$t_{offset} = t_q - t_{current}$$

Therefore, the formula for calculating the predicted position of an object is:

$$d_q = d_{stored} + v_{d_{stored}} \times t_{offset}$$

and the query that must be satisfied is:

$$d_{q_{min}} \leq d_q \leq d_{q_{max}}$$

An object in n-dimensional space qualifies a query q iff it satisfies the query in all n dimensions.

2.3.3 Range Query Precision

The precision of a range query depends on the precision of the queried data. As described in section 2.2.2, the point-based update policy provides a guarantee that objects are located within a Δ distance from the position which is stored on the server. This inaccuracy has to be taken into account when processing queries, as it affects the correctness of the results, e.g., the stored coordinates of an object might be outside the query area, while the objects actual position is inside the query area, or vice-versa. This makes it necessary to create an expansion corresponding to the Δ threshold which ensures that every object which *could* be inside the query area is included into the result. Some objects in the result may not actually be within the queried area, but since the inaccuracy is guaranteed to stay within the limits of the Δ threshold, this is acceptable.

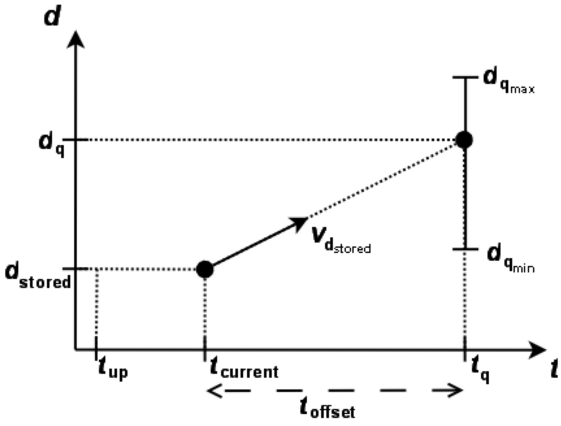


Figure 2: An illustration of how the position at query time is calculated using the current time.

One way to handle this inaccuracy in the range queries is to expand the query area in every direction by the maximum error margin of the position data. This way, all objects that are located within the original query area will be included in the result, even if their current position deviates from their stored position.

Another approach is to expand object coordinates, i.e., instead of representing objects as a points in the coordinate system, they should be represented as circles with point coordinates as the center and a radius of Δ. This way, the range queries do not need to be expanded. However, object expansions introduce another issue, which is the indexing of non-point objects. For example, R-Trees can be extended to index circles, but calculating overlaps between query rectangles and object circles requires more computation than calculating overlaps between points and rectangles. This approach is ill-suited for grid index structures, because it is impractical to index non-point objects in grids. Since query precision is not the focus of this paper, only one approach, where queries are expanded, is explored.

In the case of predictive queries, the guarantees provided for the results are tentative. When processing these queries, it is assumed that objects maintain their last reported velocities until the prediction time. The projection of object velocities into the future means that update policy guarantees only apply to the extent that this assumption holds.

2.3.4 kNN Query Semantics

A standard kNN query is defined like this: find k nearest neighbours (objects) for a point q. The parameter k defines how many nearest neighbours to find, whereas the parameter q defines the query point. The semantics of kNN queries are as fol-

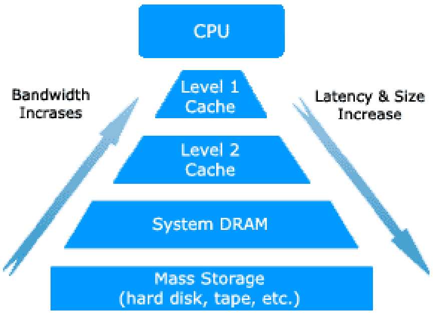


Figure 3: Overview of hardware memory hierarchy architecture.

lows. Given a query point q on a set of moving objects O, it must be ensured that the query result set O', which is a subset of O, satisfies the conditions

a)$|O'| = k$
b)$\forall o \in (O - O'), dst(q, o) \geq Max\{dst(q, o')|o' \in O'\}$

Condition (a) ensures that the query result set contains k objects, whereas condition (b) ensures that these k objects are the k nearest ones to q. In condition (b), $dst(q, o)$ is the distance betweeen the query point q and the location of object o, and $dst(q, o')$ is the distance between query point q and object o'.

2.4 main-memory

Traditional database applications require a high level of persistence. Hard-disk drives are persistent and have therefore been the preferred medium, and thus DBMSs are optimized for storing data on disks. Users of applications for monitoring moving objects, however, tend to value speed over persistence, e.g. historical information about the 10 closest emergency vehicles two weeks ago, is not important. Instead, users want fast and accurate results of queries on the current and predicted positions of moving objects.

main-memory sizes have increased in recent years, and since the cost of accessing disk is orders of magnitude slower than accessing main-memory, implementing the database in main-memory is now an option that can boost the speed of applications.

Running the database in main-memory does not eliminate performance bottlenecks related to the hardware. The problems rather move up one level in the memory hierarchy [8]. In disk-based databases, indexing structures are optimized for loading disk-blocks consecutively into main-memory, in order to avoid random accesses which are slow compared to the main-memory and become a bottleneck since the CPU must idle while fetching additional data from disk.

When the database runs in main-memory, data access is much faster. However, main-memory speeds are significantly slower than CPU, and the performance gap between CPU and main-memory is steadily increasing [2]. The current development suggests that the ratio of CPU instructions to random main-memory access will soon be up to 1:1000. That is, CPUs perform 1000 cycles between random accesses to main-memory [4].

To narrow this gap, the size of the Level 1 and Level 2 CPU caches (see Figure 3) have been increased by hardware manufacturers, in order to minimize CPU idle time. These memory caches operate at speeds that are closer to the CPU. However, the cache memory is very expensive, so the size is still limited.

Unlike disk drives and main-memory, the OS and programmers have no control over the CPU caches, as that would incur a heavy resource administration overhead, which would eliminate the gains of having the caches [4]. The caches are therefore administered by the CPU. The CPU will attempt to optimize cache utilization, such that the number of random main-memory accesses (or cache misses) is minimized, using concepts like spatial and temporal locality.

This means that main-memory applications can be designed to be *cache conscious*. Even though we do not have direct control over CPU caches, we can still implement applications that take the memory hierarchy structure into account, and e.g., use arrays instead of linked lists to preserve spatial locality, such that data can be prefetched from lower levels in the memory hierarchy, making the application perform faster.

Disk-based applications are typically optimized for disk blocks. Similarly, main-memory applications can be optimized for the cache lines. Since the cache line size is typically no more than 64 or 128 Bytes, corresponding to a mere 3-5 objects of size 20-40 Bytes, the optimal size of data elements is usually some multiple of cache lines.

Our proposed designs follow the above mentioned guidelines, e.g. data locality is preserved. Having described the setting, we proceed to discuss the related work in Section 3. In Section 4, we consider various approaches that provide solutions to the mentioned issues of updating and querying main-memory indexes.

3 Related Work

The research in spatial database indexes has traditionally focused on Geographic Information Systems (GIS) that store few, complex and relatively static spatial objects like polylines or polygons with large numbers of vertices and which perform few, advanced queries with expensive spatial joins [14]. This is evident from the design of traditional spatial indexes like the R-tree proposed by Guttman [5], which is optimized to handle queries efficiently. This comes at the cost of update efficiency.

Many new applications monitor large numbers of simple, dynamic spatial objects, e.g. points that are frequently updated and perform frequent albeit relatively simple queries, e.g., range queries. Thus, there is a need for indexing structures which efficiently handle vast amounts of updates, in addition to frequent queries.

In Section 2.2 we introduced update policies, which are used to reduce the amount of updates. Wolfson et al. [16] proposed dead-reckoning policies, which use a deviation threshold to determine when to update the position of an object. This is also the concept of shared-prediction updates proposed by Civilis et al. [3]. Dead-reckoning policy requires objects to store road network information and possible predefined routes, whereas in point-based and vector-based shared-prediction update policies, this is not neccesary. We use the point-based update policy, because it requires the least amount of additional calculations in the indexes.

Several indexes for dynamic moving object tracking have been proposed, e.g. tree structures like the *TPR-tree* [15], *B^x-tree* [6], *B^{dual}-tree* [19], and grid-based structures like the *LUGrid* [18] which are all disk-based.

Some LBSs require information about past positions, whereas other applications only index current and near-future positions of objects. Persistence is required in the first category of applications, and is well supported in disk-based implementations, but it is typically less important in the second category of applications. Increasing main-memory sizes make it feasible to implement databases in main-memory, and since main-memory access is much faster than disk access, this significantly reduces the update and query response time of an application.

Indexes like the cache sensitive *B*+-tree, or *CSB+-tree*, proposed by Rao et al. [12] and the cache conscious R-tree, or *CR-tree*, proposed by Kim et al. [8], optimize the design of their disk-based counterparts for better cache utilization. main-memory implementations are typically optimized for cache line size, which is much smaller than the size of a disk block. The *CSB+-tree* therefore uses pointer-elimination, where nodes on each level of the tree are stored contiguously, such that more node data can fit in each cache line. In the *CR-tree*, this technique has less impact due to the higher space consumption of each node, and therefore the *CR-tree* also applies techiques like quantization and key compression, storing MBR coordinates relative to parent MBRs. In both approaches the aim is to compress the data stored in the indexes, which adds addidional cost to updates and queries. To some extent, data compression can be added to any index. Therefore for the relative comparission of indexes it does not have a significant influence and we do not explore it further.

Authors of spatio-temporal indexes typically compare their index to one or a few other competitive indexes, e.g. [12] and [8]. However, these comparisons tend to focus on the promising properties of the proposed index. Jensen et al. [7] propose the COST benchmark for evaluating disk-based indexes for the current and near-future positions of moving objects, measuring CPU and I/O performance of queries and updates. Myllymaki et al. propose the DynaMark benchmark [11] for evaluating main-memory indexes, but they do not consider future predicted positions of objects.

4 Solution Overview

First, we give an overview of well-known data structures that can be used for indexing moving objects. Second, the advantage of adding a secondary index is discussed.

4.1 Data- And Space- Partitioning Indexes

As discussed earlier, LBSs are becoming increasingly popular, so the number of users is expected to rise. Hence, the data structure used to implement applications must be scalable.

Linked lists provide quick insert operations, and allow the number of users to grow without increasing costs of insertion. However, the update performance scalability of a linked list is inadequate since, in the average case, half of the list has to be traversed to locate an object. Alternatively, an array or a hash table structure where each object has a unique ID, which is used for indexing, provides fast updates, but both structures need to be reallocated as the number of objects increases.

In addition, a key requirement is that the monitoring applications must support fast queries (e.g. current-time and predictive range queries, as well as k-NearestNeighbour queries). However, in each of the mentioned data structures, a full traversal is needed in order to determine the result of each query. If the number of monitored objects is large, e.g., 100,000, and the size of the queries correspond only to a small fraction of the monitored area, then queries need to be performed more efficiently.

The R-tree structure [5] performs fast queries on static, multi-dimensional data. However, R-trees are slow at performing updates due to the cost of maintaining the tree structure.

This scenario has inspired the research of indexing structures that support fast updates *as well as* fast queries. The widely used B^+-tree index structure generally supports these requirements, but does not support multi-dimensional data.

Structures like the B^x tree [6] uses linerization, space-filling curves and multiple B^+-trees to represent multi-dimensional data. Despite the linerization techniques, handling multiple dimensions using B-trees remains fairly complex and incurs a significant computational overhead. Though space-filling curves provide efficient range query processing, the support for kNN queries is poor.

In the previous paragraphs we only discussed structures which index the data. An alternative approach is to index the monitored space. This is typically done using a multi-dimensional array, or *grid*, where each element, or *cell*, represents some spatial range of the monitored area. One example of such a structure is the LUGrid [18], which performs delete and insert operations *lazily* to optimize update performance. The direct mapping from coordinates to the address of a grid cell allows fast updating. However, grid performance degrades when most objects are concentrated in one or a few grid cells (position skew), since all objects in a cell need to be traversed during updating and querying. Moreover, memory has to be allocated for *all* cells, even if many of them contain no objects.

Some Quad-trees also index the monitored space, but unlike the grid, a quadtree partitions the space dynamically. In quad-trees the nodes are always partitioned into four children when their capacity is exceeded; one for each quadrant in the 2 dimensions, divided on the North-South and East-West axes. The quad-tree therefore tends to partition the space where objects are concentrated, making it better suited for skewed data, and does not suffer from the drawbacks of the grid. However, the quadtree is not a balanced structure, making worst-case query performance worse than that of a linear scan of the dataset, which is inadequate.

4.2 Secondary index

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 updates, the outdated object data must be located using spatial information.

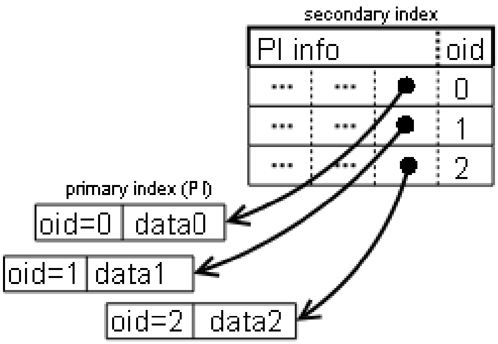


Figure 4: Abstract example of a secondary index.

This procedure is cumbersome, and if all objects have a unique object ID *oid*, there is a more efficient approach. The idea is to create a secondary index, which uses *oid* as the index key and points to the location in the primary index associated with that key. In addition, the data of the secondary index can include extra primary index information for an object, e.g. an index number in an array. The concept is illustrated as an abstract example in Figure 4. The secondary index uses *oid* as its key, and its data is the primary index information (PI info). Part of that information is a pointer to the primary index, where the data of an object is located. This way, near constant O(1) updating time can be achieved, whereas when using spatial data for updating, constant O(1) time is hardly achievable.

A hash table is a suitable structure for secondary indexes, as it supports efficient insertions, deletions and searches. Although the time spent in searching depends on the hash function and the load on the hash table, both insertions and searches approach O(1) time, especially when keys are unique, which is the case with *oid*s.

The efficiency of an indexing structure is often a tradeoff between update and query performance. We propose indexing structures that consist of two indexes, a primary index ensuring efficient queries, and a secondary hash table index which provides fast updates. The processing costs of maintaining the secondary index are compensated by faster object look-ups.

Instead of examining advanced indexing structures, we focus on examining the performance of basic space and data partitioning indexes, that is, a Grid and an R-tree implementation, which we extend by using secondary indexes and optimize for running in main-memory.

5 Structures

In the following two subsections, we discuss the design and optimizations of updating and querying in our proposed main-memory variants of Grid and R-tree index structures and their secondary indexes. In the third subsection, we present a simple array design, which is the naïve solution to indexing, and discuss the pros and cons of that solution.

5.1 Grid

The grid is a space partitioning index, where a predefined monitored area is divided into equal and fixed size rectangles, referred to as *cells*. Objects, whose coordinates fall within the boundaries of a grid cell belong to that particular grid cell. Each grid cell holds a list of objects that are within the boundaries of that cell.

The main parameters defining the grid are the total monitored area and the grid cell sizes, x_{gcs} and y_{gcs}, respectively for the x and y dimensions. x_{gcs} and y_{gcs} define the lengths of a cell in the respective dimensions.

Formally, objects are within the same grid cell if their x coordinate is within the range $[x_{cell}, x_{cell} + x_{gcs})$ and their y coordinate is within the range $[y_{cell}, y_{cell} + y_{gcs})$, where $x_{cell} = \lfloor \frac{x}{x_{gcs}} \rfloor \cdot x_{gcs}$ and $y_{cell} = \lfloor \frac{y}{y_{gcs}} \rfloor \cdot y_{gcs}$.

5.1.1 Design

The overall design of the grid structure is illustrated in Figure 5. The structure consists of a grid index and a secondary index.

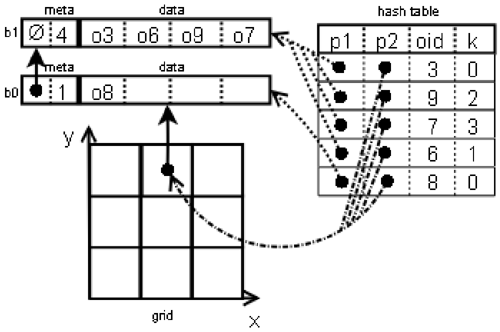


Figure 5: An example of the overall design of the grid structure

Since array structures are well-suited for organizing data efficiently in main-memory [1], the grid index is stored as a two-dimensional array of pointers. Each grid cell within the array contains a pointer to the linked list of buckets. In the example in Figure 5, $b0$ and $b1$ are the first two buckets of a grid cell. Objects within a grid cell are grouped into those buckets, which are of a fixed and predefined size. The grouping has no spatial or any other type of ordering. Objects are stored in buckets, because the data to be processed during updating or querying is loaded in blocks (cache lines) to the CPU cache from main-memory as mentioned in Section 2.4. A consecutive block of memory is allocated for each bucket. The size of a bucket is called a *bucket_size* and is defined in bytes. Each bucket has data and meta data fields. The meta data field contains a pointer *next*, which points to the next bucket in the list of a grid cell and k, which is the current number of objects in the data field of the bucket. The data field contains the object data (o3, o6, o7, o8, o9 in Figure 5). As defined in Section 2.1 object data consists of an *oid*, x, y, v_x and v_y.

Every object in the primary index has exactly one entry in the secondary index. The entry consists of four fields $p1$, $p2$, *oid* and k. The pointer $p1$ points to the bucket of a grid cell that contains the object, whereas the pointer $p2$ points to the grid cell that contains the bucket, where the object data is located. The *oid* is used as a unique hash table key for fast look-ups. The last parameter k is an index number which is used to determine where exactly the data of an object is stored in the bucket.

As an example of the composition of a bucket, in a 64-bit system the size of an object data is $8(oid) + 4(x) + 4(y) + 4(v_x) + 4(v_y) = 24$ bytes. The meta data field of a bucket takes up 16 bytes of memory, 8 bytes for the *next* pointer, 4 bytes for the k and 4 bytes of padding, which should not be discarded, since the pointers and *oid*s have to be aligned on 8 byte boundary in main-memory as discussed later in Section 6.2. The data field takes up $(bucket\_size - 16)$ bytes. So the maximum number of objects k_{max} that fits into a bucket is calculated using this formula: $k_{max} = \lfloor \frac{bucket\_size - 16}{24} \rfloor$.

5.1.2 Updating

As objects change position, the index needs to be updated accordingly. An object might move from one grid cell to another, triggering a *non-local* update, or stay within the boundaries of the current cell, triggering a *local* update. If an update is local, then only the values of coordinates x and y and velocity vectors v_x and v_y are updated. If an update is non-local, the object must be deleted from the old cell, and inserted into the new one.

The design is optimized for local updates, where only the object data, i.e. x, y, v_x, and v_y is updated (overwritten). The optimization is achieved by using the primary index information in the hash table, namely the bucket pointer $p1$ and the object position k within a bucket. In a local update, $p1$ and k are used to compute the address of the data to be updated in memory, hence no scanning is required, which is particularly desirable in grids with large grid cell sizes. After finding the memory address, the data is overwritten. In the delete operation, a linear scan of buckets and a linear scan within a bucket, to locate the object, are also avoided by calculating the memory address of the object to be deleted using $p1$ and k. Although maintaining $p1$ and k adds additional computational cost during non-local updates, it is compensated by the gain of making look-ups instead of linear scans during local updates and delete operations.

The pointer $p2$ from the hash table entry is used to efficiently determine whether an object has moved outside the boundaries of its current grid cell. It is also used during delete operations, when moving the last object from the first bucket of a grid cell into the bucket space of the deleted object. This ensures that all except one bucket per grid cell are full, which enhances query performance.

The algorithm in Table 1 shows how updating the grid structure is accomplished. The algorithm checks if the object being updated already exists in the hash table and creates it if it does not (Lines 3-7). The pointer *hte* is a hash table entry of an object. If the coordinates of the update are within the grid boundaries (Line 8), the new cell is determined

```
 1  void update(Update upd){
 2    pointer hte, new_cell, obj;
 3    hte = ht_find(upd.id);
 4    if (hte = NULL){
 5      hte = allocate_memory();
 6      ht_add(upd.oid, hte);
 7    }
 8    if (upd is within grid boundaries){
 9      new_cell = getcell(upd.x, upd.y);
10      if (new_cell != hte->p2){
11        if (hte->p1 != NULL)
12          delete(hte);
13        insert(new_cell, hte, upd);
14      } else {
15        offset = hte->k * size(objectdata)
16        obj = hte->p1 + size(meta) + offset;
17        obj->[x,y,vx,vy] = upd.[x,y,vx,vy];
18      }
19    } else
20      if (hte->p1 != NULL)
21        delete(hte);
22  }
```

Table 1: The update operation of the Grid.

and checked against the old cell of the object (Lines 9-10). The *getcell*(x, y) operation (Line 9), returns the pointer to the grid cell corresponding to the coordinates x and y. If the object moved from its old grid cell, it is deleted from that cell and inserted into the new one (Lines 11-13), otherwise the data of the object is located within the grid index and only the values are updated (14-18). If the coordinates of the update fall outside of the predefined grid area, the object is deleted (Lines 19-22). The deletion (Line 12) and insertion (Line 13) algorithms are shortly described in the following two paragraphs.

The delete operation, which is shown in Table 2, is a simple and efficient operation done in near constant ($O(1)$) time. The operation is supplied with a hash table entry pointer, *hte*, of the object to be deleted. Local operation variables are created and assigned (Lines 2-8). First it is checked whether the bucket is the first in the linked list of the grid cell (Line 9) and if the object to be deleted is not the last element in the bucket (Line 10). If both conditions are true, the last element of the bucket is moved into the place of the object to be deleted (Lines 11-13). If the bucket is not the first in the linked list, then the last element is taken from the first bucket and inserted into the place of the object to be deleted (Lines 15-21). Finally the element count of the bucket is decreased by 1 (Line 22). If the bucket becomes empty, it is removed from the grid cell (Lines 23-26), and the hash table entry of the deleted object is updated (Lines 27-28).

The insert operation, which is shown in Table 3, is also performed in near constant ($O(1)$) time. Local variables are created and assigned in (Lines 2-4). First it is checked whether there are no buckets in the cell or the first bucket is full (Line 5). If so, a new bucket is created and inserted (Lines 6-8), if not, the *insert_at* is set (Line 10). Finally the hash table is updated (Lines 12-14) and the object data is inserted (Lines 15-16).

```
 1  void delete(pointer hte){
 2    pointer first_bucket, bucket;
 3    pointer first_data, data, hte_last;
 4    int last_k;
 5    first_bucket = hte->p2->next;
 6    bucket = hte->p1;
 7    data = bucket + size(meta);
 8    last_k = first_bucket->k - 1;
 9    if (bucket == first_bucket){
10      if (hte->k != last_k){
11        hte_last = ht_find(data[last_k].oid);
12        data[hte->k] = data[last_k];
13        hte_last->k = hte->k;
14      }
15    } else {
16      first_data = first_bucket + size(meta);
17      hte_last =
            ht_find(first_data[last_k].oid);
18      data[hte->k] = first_data[last_k];
19      hte_last->k = hte->k;
20      hte_last->p1 = hte->p1;
21    }
22    first_bucket->k--;
23    if (first_bucket->k == 0){
24      hte_last->p2->next =
            first_bucket->next;
25      free(first_bucket);
26    }
27    hte->p1 = NULL; hte->p2 = NULL;
28    hte->k = -1;
29  }
```

Table 2: The delete operation of the Grid.

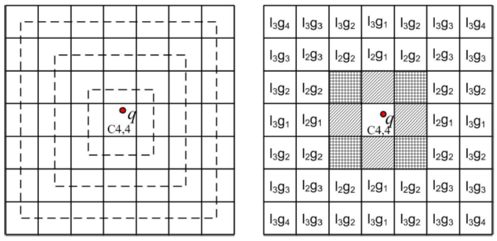
```
 1  void insert(new_cell, hte, upd);{
 2    pointer bucket = new_cell->next;
 3    pointer data;
 4    int insert_at = 0;
 5    if (bucket == NULL || bucket->k ==
          k_max){
 6      bucket = allocate_memory();
 7      bucket->next = new_cell->next
 8      new_cell->next = bucket;
 9    } else
10      insert_at = bucket->k;
11    data = bucket + size(meta);
12    hte->p1 = bucket;
13    hte->p2 = new_cell;
14    hte->k = insert_at;
15    data[insert_at] = upd;
16    bucket->k++;
17  }
```

Table 3: The insert operation of the Grid.

The insertion and deletion algorithms ensure that only the first bucket of a grid cell can be not full, whereas all the other following buckets are full. By having a large amount of fully filled buckets, the number of buckets needed to hold all the objects is decreased, which results in better query performance.

5.1.3 Parameters

The performance of the Grid is adjusted by tuning the defining grid cell size $gcs = x_{gcs} = y_{gcs}$ and *bucket_size* parameters. If the *gcs* is very low, e.g. a small fraction of the extent in a given dimension, then objects change grid cells more often and the average cost of an update becomes higher.


q
C4,4
l3g4 l3g3 l3g2 l3g1 l3g2 l3g3 l3g4
l3g3 l2g3 l2g2 l2g1 l2g2 l2g3 l3g3
l3g2 l2g2 l2g2 l3g2
l3g1 l2g1 C4,4 q l2g1 l3g1
l3g2 l2g2 l2g2 l3g2
l3g3 l2g3 l2g2 l2g1 l2g2 l2g3 l3g3
l3g4 l3g3 l3g2 l3g1 l3g2 l3g3 l3g4

(a) Layers (b) Groups

Figure 6: kNN query cell groups.

Range queries are also affected by this parameter, depending on their coverage area, e.g., if a range query coverage is very low, then the Grid with a low *gcs* performs better than the Grid with a high *gcs*. Increasing the *bucket_size* also increases the query performance due to a better storage locality of objects within the cell. Updates are not affected significantly by the different values of the *bucket_size*, because the exact memory location of an object is determined by using the secondary index, i.e. the hash table. The last parameter of the Grid is the total monitored area, which is a rectangle that defines the scope of the Grid using two boundary points.

5.1.4 Range queries

A current-time range query is defined by two points in the coordinate system $(x_{q_{min}}, y_{q_{min}})$ and $(x_{q_{max}}, y_{q_{max}})$. The algorithm is as follows. First, the cells covered by the range query are split into two groups - fully covered and partially covered. The objects from fully covered cells are put into the result list by scanning the linked lists of buckets. Objects from partially covered cells are checked individually to determine whether they are within the range of the query and only then are they put into the result list.

A predictive range query is defined by two points in the coordinate system $(x_{q_{min}}, y_{q_{min}})$, $(x_{q_{max}}, y_{q_{max}})$ and a prediction time offset t_{offset}. The algorithm is similiar to the current-time range query. It is done by expanding the original range query by $exp = v_{max} \times t_{offset}$ in every direction in every dimension. Afterwards all the linked lists of the cells that are partially and fully covered by the expanded range query are scanned through and all objects are checked to determine whether they will be within the original range of the query after the t_{offset}. Formally an object whose coordinates are x and y, velocity vectors are v_x and v_y satisfies the query if:

$x_{q_{min}} \leq x + exp \leq x_{q_{max}}$
and
$y_{q_{min}} \leq y + exp \leq y_{q_{max}}$

5.1.5 kNN queries

A kNN query is defined by a query point q and the number of nearest neighbors required k. The algorithm used for performing kNN queries in Grids is fully described in a recent paper by Wu and Tan [17]. The main part of the algorithm is called VOB (Visit Order Builder). The cells in the grid index are divided into different groups, such that cells within each group have similiar minimum distances to the query point. The division is illustrated in Figure 6. The point q is the kNN query point and $cell(q)$ is the grid cell where the query is. First the grid cells around $cell(q)$ are divided into different levels. The cells around $cell(q)$ form the first level. The cells around level l form the $(l + 1)$ level. In Figure 6(a) cells connected by a dotted line belong to the same level. $cell(q)$ is defined as level 0, and level $l + 1$ is defined as the cells around level l. Next the cells in each level l are divided into $l + 1$ groups based on their relative positions to $cell(q)$ as illustrated in Figure 6(b). Each group has 4 or 8 cells depending on the position of the group. Cells within a group have similiar minimum distances to the query point. L_lG_g is used to refer to the Group g of Level l. The formed groups have the following properties:

- Each cell belongs to one and only one group.
- The cells in the same group have similar minimum distances to q. (This property does not hold for small values of l.)
- The minimum distance between L_lG_g and q is smaller than the minimum distance between $L_{l+1}G_g$ and q.
- The minimum distance between L_lG_g and q is smaller than the minimum distance between L_lG_{g+1} and q.

A priority queue, where the elements are ordered by their minimum distance to q, is formed. The priority queue is implemented using a heap data structure. Heap data structure is a good choice for implementing a priority queue, because, given a priority queue of n elements, insert and delete operations are done in $O(log_2(n))$ time, whereas finding the element with the lowest minimum distance is done in $O(1)$ time. The element pushed into the priority queue is either a cell or a group. Initially, the cell $cell(q)$ and group L_1G_1 are pushed into the priority queue. A function $NextCell(q)$ de-queues an element from the priority queue. If a cell is de-queued, it is just returned. If a group L_lG_g is de-queued:

- The cells within the group (computed on-the-fly) are pushed into the priority queue;
- Group $L_{l+1}G_g$ is pushed into the priority queue;
- If $g = l$, then L_lG_{g+1} is pushed to the priority queue.

```
1  void kNN_query(Point q, int k){
2    int k_found = 0, min_dist = 0;
3    int cd = MAX_INT_VALUE;
4    pointer cell = NULL;
5    OrderedList rl;
6    PriorityQueue pq;
7    pq->enqueue(cell(q));
8    pq->enqueue(L1G1);
9    while(cd > min_dist){
10       while (cell == NULL)
11         cell = NextCell(q);
12       min_dist = cell->min_dist(q);
13       for (Every object in the grid cell)
14         if (k_found < k){
15           rl->enqueue(object);
16           k_found++;
17         } else
18           if (object->min_dist(q) <
                  rl->max_dist(q)){
19             rl->ReplaceFirstElementWith(object);
20             cd = object->min_dist(q);
21           }
22   }
23   return rl;
24 }
```

Table 4: The kNN operation of the Grid.

- Recursion to $NextCell(q)$ is used until a cell is returned.

In the original description of the kNN algorithm, when a group is de-queued, recursion is used until a cell is de-queued and returned. Depending on the data, the grid cell size and the kNN query parameters the algorithm can end up doing an excessively deep recursion, which can overflow the call stack and result in the crash of the application as explained in Appendix B. In our design, the recursive calls are substituted with a cycle in the kNN query algorithm, therefore the last item in the above list is not performed.

A high level description of the kNN query operation is shown in Table 4. The operation is supplied with a query point q and the number k. The critical distance parameter *cd* is assigned a maxmimum integer value (Line 3). In Lines 2-4 initial parameters are created and set. The priority queue *pq* is ordered ascendingly by the minimum distance to the query point q. The result list *rl* stores the k nearest objects to point q, and is ordered descendingly by the minimum distance to the query point q. It is ordered descendingly, because when there are k items in the list, only the first item needs to be checked and replaced if neccessary. Initially, *cell*(q) and L_1G_1 are enqueued into the priority list (Lines 6-7).

The *while* cycle (Line 8) is continued until the *cd* is higher than the *min_dist* of the last visited cell. First the next nearest cell to q is acquired (Lines 9-10). Then all the objects within that cell are checked (Lines 12-20), and if k objects are already found, they are checked against the first object (the one with the highest *min_dist* to q) of the result list *rl* and replaced if neccesary (Lines 17-20). If less than k objects are found, the object is inserted into the result list *rl* (Lines 13-15).

5.2 R-Tree

The R-tree is a balanced, tree based, data partitioning index structure for indexing multi-dimensional objects. It is based on the concept of minimum bounding rectangles (MBRs). An MBR is the smallest rectangle that encloses a group of point objects, and in the R-tree a node is defined by an MBR which encloses all of the descendants of the subtree. The R-tree is a data partitioning index because it divides children in a node based on their relative locations without considering the underlying space in which the tree is defined. Depending on the design of the tree, MBRs within one level of the tree can be allowed to overlap, making the insertions and updates less computationally heavy than if the nodes had strictly non-overlapping MBRs. This is a trade-off between update performance and query performance, since overlapping MBRs might necessitate investigating several branches of the tree that cover the same area. The details of this issue is discussed later in Section 5.2.2.

The R-tree has two parameters that affect the growth and balancing of the tree structure; the *fanout* and the *minchildren*. The *fanout* defines the maximum number of children that a node can contain. The *node_size* is the size of allocated memory for each node. This consists of the node meta data and the $fanout$ multiplied by the size of a child. If a node exceeds the *fanout*, a split occurs and the children are divided between the overfull node and a new node which is added to the parent node. If a split occurs in the root node, the height of the tree is increased by one level, creating a new root node and inserting the old root and its new sibling. This is also the reason why the tree is balanced, since the tree grows upwards and all objects are inserted at the lowest level of the tree. The lower the fanout is the higher the tree becomes because more leaf nodes are required to contain the objects and more internal nodes are required to contain those leaf nodes. However, if the fanout is high the nodes themselves become larger, taking up more space and costing more to process individually. The higher number of children in a node also means that each node covers a larger area and thus searches become more inefficient because it is more likely that a child of the node that is being processed is not within the query area. This means that there is a trade-off between the costs of traversing the tree and the costs of processing large nodes.

When the R-tree is used to index static data, it grows as new data is inserted, but when used to index dynamic data, the tree can also shrink when a child is deleted from a node, or make a branch of the tree shrink if a child is moved from one node to another. This makes it necessary to have another parameter, *minchildren*, defining the minimum number of children a node may contain before it should be merged with other nodes. *Minchildren* can be any integer value between 1 and *fanout*/2. The reason why *minchildren* can not be higher, is that a split cannot divide the children of a node if more than half are required to be in each of the resulting nodes, since the split would then immediately trigger a merge due to the required *minchildren*.

5.2.1 Design

We propose a main-memory R-tree which is composed of nodes that each contain meta data and information about their children. Each node is assigned to a continuous part of memory in order to maintain data locality for faster access. As illustrated in Figure 7, there are two types of nodes in the index; internal nodes, which are nodes containing other nodes and leaf nodes, which are nodes at the lowest level of the tree and contain the objects stored in the R-tree, i.e. the leaves of the tree structure.

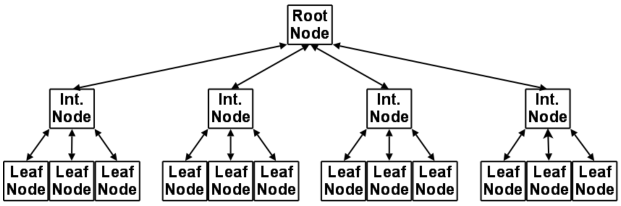


Figure 7: R-tree overall structure.

As illustrated in Figure 8, an internal node is composed of a number of child entries followed by node meta data describing aspects of the node itself. Each child entry describes the location of the child and the MBR that encloses the objects within that child. In this way, it is possible to get the information about all children of a given internal node without having to access each of them individually, which would produce an overhead in terms of time consuming node accesses.

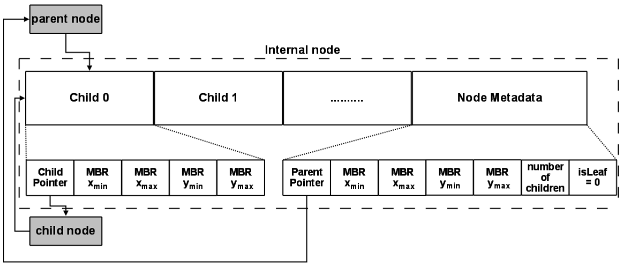


Figure 8: R-tree internal node composition.

The node meta data information can be divided into four parts, detailing the location of the node parent, the MBR of the node, the number of children in the node, and a variable indicating whether the node is a leaf node or an internal node. The location of the parent node and the MBR are both used when traversing up the tree. This will be discussed in detail in the following section. The 'number of children'-value is stored to support efficient delete and insert operations within a node. During inserts, the value is used to calculate the offset to the first available element, such that objects can be inserted in constant time, without having to scan the array. The last variable is used to indicate whether the node is a leaf node. Although the size of internal nodes and leaf nodes is the same, leaf nodes do not store the same information in the child entries. The root node shown in Figure 7 is the same as an internal node, the only difference being that its parent node variable is empty.

The child entries are stored before the node meta data because of the way accessing entries is done using offsets. Since child entries are the same size as the first two parts of the node meta data, it is possible to offset into the node in increments of that size and get to the next child. The node MBR and parent pointer can also be accessed easily by offsetting into the node by the *fanout* value.

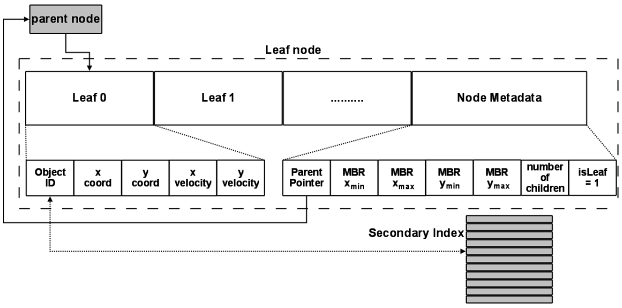


Figure 9: R-tree leaf node composition.

Leaf nodes contain the objects stored in the tree. As illustrated in Figure 9, the information stored in the leaf nodes differs from their internal node counterparts, in that the children are point objects with a velocity vector. Another difference is that the objects are stored in the node, so pointing to the child location is unnecessary. Instead, the R-tree stores the ID of the object.

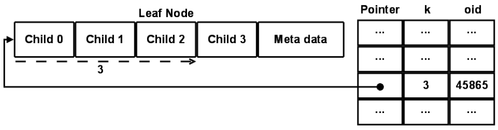


Figure 10: An example of the secondary index in the R-Tree

The secondary index entries consist of pointers to leaf nodes, in which the objects are stored. They also contain the array index numbers of objects in the leaf nodes. This way, it is possible to find a specific object location without searching the primary index. As illustrated in Figure 10 the object entry in the secondary index has a pointer to the leaf node where the object is stored and a k value that defines the offset in that node.

5.2.2 Updating

An R-tree is updated by locating and deleting/reinserting the updated object. In the original R-tree [5], this process is expensive in terms of processing and accessing data, but if the data is mostly static[1] and advanced queries with many spatial joins have to be performed efficiently, the high update costs represent an acceptable trade-off. Since we expect a high number of updates for dynamic, spatio-temporal objects, it is necessary to process updates faster.

When performing a split in the R-tree, three algorithms are proposed by Guttman [5]:

The Exhaustive algorithm creates the best possible split based on area and overlap of the resulting nodes. However, this algorithm runs in exponential time and it is therefore unsuited for our implementation where fast update processing is essential.

Quadratic-Cost algorithm attempts to find a small-area split but is not guaranteed to find the smallest one. It runs in polynomial time. Since the goal is to handle large amounts of updates this is also too processor heavy.

Linear-Cost algorithm identifies the two child nodes that are furthest apart relative to the node MBR of the splitting node. After the seeds for the split have been chosen, the rest of the children are divided between the nodes according to the area of enlargement necessary to add the child to a given node.

We use the *Linear-Cost* algorithm. Despite its ability to provide fast node splits, it is not the most efficient algorithm for queries, since the MBRs of siblings are allowed to overlap each other, whereas the other algorithms are designed to minimize or avoid overlapping MBRs. The reason for choosing this algorithm is that it works in linear time which suits our goal of providing an index that perform fast updates.

When the data is dynamic, accessing and processing costs must be kept at a minimum. Using the idea from Lee et al. [10], the goal is to perform updates from the bottom up, making it possible to avoid traverses down the tree. As explained above, the update process can be divided into two operations:

Locating: Instead of using the old coordinates of an object to find its position in the tree we use the secondary index. This way the object is located from the bottom without having to traverse the tree.

Deleting/Reinserting: Instead of always performing a deletion and insertion when updating an object, we check if the new location of the object is still within the MBR of the old node. Otherwise we traverse up the tree until we find a node which the object is still inside and then insert from there. This update strategy has a worst case scenario where the tree is traversed from leaf to root before a suitable subtree is found, but in the average case, objects will not change their position radically from one update to another, and thus the objects will tend to change location locally within the subtree.

5.2.3 Range Queries

In the R-tree, range queries are performed by checking from the root of the tree to the branches if a query area overlaps the MBRs of the children. It is a recursive function that is called on the children of a node. The principle is shown in Figure 11 where the query area is overlapping the node MBR. Then the children of the node (sub-nodes A-C) are checked to determine whether their MBR overlaps the query area. The MBRs of sub-nodes B and C overlap so they are, in turn, checked to determine whether their children overlap the query area. The search of the tree is performed depth-first, such that the child of a qualifying node will be checked before the siblings. When the search reaches a leaf node all children in that node are checked and the ones that are located in the query area are added to the results. As explained in section 2.3, the only difference between a current time range query and a predictive range query is that the query area is first expanded by the offset time and the objects that qualify in the expanded query area are checked to determine whether their predicted positions are within the original query area.

5.2.4 kNN Queries

The kNN queries are performed using a heap structure and a linked list which is referred to as the Active Branch List (ABL), and the result list, respectively. These are used for intermediate node results and object results, respectively. The way a kNN query is processed is shown in Figure 12 and it is based on the design by Kuan et al. [9]. That article uses the techniques developed by Roussopoulos et al. [13], but concludes that with clustered data the

[1] e.g. land register data, where it can be expected that the data is only rarely changed.

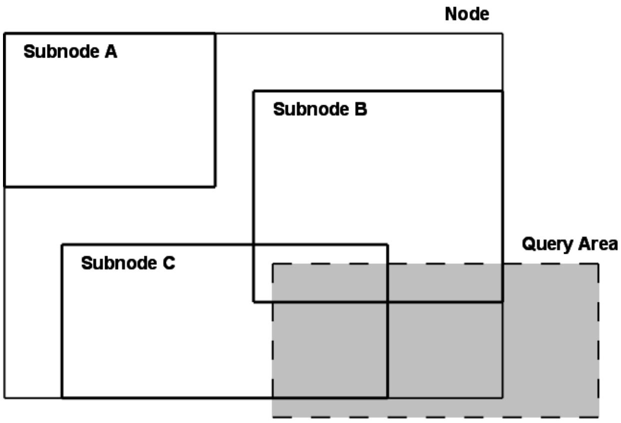


Figure 11: R-tree overlapping range query

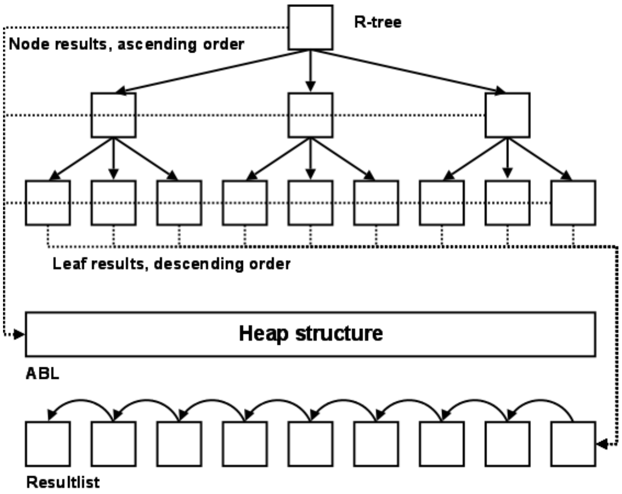


Figure 12: R-tree kNN query processing.

kNN query can be performed much faster without calculating the MinMax distance. Kuan et al. also conclude that even without clustered data, the kNN query is marginally faster when measuring only the minimum distance to a node MBR and it requires less computation to acquire the result. This suits the design criteria of minimal processing and fast performance that we require when implementing the kNN query in the main-memory R-tree.

The ABL is a distance prioritized heap of nodes awaiting processing. Elements of the heap consist of the location of a node and the minimum distance from that node to the query point q. If q is inside a node MBR the distance is 0, since a negative distance is not possible.

When an element of the ABL is processed, the distance to the children of the corresponding node are inserted in the ABL and the processed element is discarded. Thus the ABL works as a priority queue where the closest element is processed and discarded.

If the element is a leaf node, the children of the node are objects, which are therefore inserted in the result list instead of the ABL. Elements of the result list contain object data instead of the locations of nodes, but otherwise they are identical to elements of the ABL. The ordering of the result list is descending, as the objects that are inserted into the list originate from nodes that are processed in ascending order. This means that the objects are on average going to be further away from q, the further up the ABL they are situated. Ordering the result list in descending order means that it is not necessary to traverse to the end of the linked list for every new result. The number of objects in the result list is counted and when k results have been found, additional results are compared with the highest stored result and the highest of the two results is discarded.

The kNN query continues until the highest recorded result is lower than the lowest remaining ABL element, since any objects within that element will have the same or higher distance to q.

5.3 Array

The Array is an index which does not partition the contained elements. The order of the elements is not related to their spatial location, instead they are indexed by their *oid*s, and stored sequentially in main-memory which allows for fast insert, delete and update operations.

Since there is no spatial ordering of the elements, it is necessary to traverse the entire structure in order to answer spatio-temporal queries, i.e., this structure is used to examine to which extent fast updates can compensate for slow queries, to determine the level of justification, if any, of advanced indexing structures like the Grid and the R-tree.

One major disadvantage of the array structure is that it needs to be rebuilt in the event that the number of monitored objects exceeds the initially defined size of the array. This leads to a severe degradation of performance during the regeneration process.

6 Implementation

Before we proceed to describe our experimental setting, we want to discuss some challenges that apply specifically to main-memory implementations. Beyond the discussions within this section, some other implementation details are presented in Appendices A, C, D, and E.

6.1 Timing measurements

Since the indexes run in main-memory, operations are performed very fast compared to disk-based indexes. Therefore, the Standard C++ Library *clock*() function, which depending on the operating system can measure up to 1 micro second granularity, is not sufficiently accurate.

Today, most CPUs have an internal cycle

counter, which counts the total number of CPU cycles from the computer boot up. The RDTSC instruction is used for reading the value of the counter. The instruction is used to measure the cycle count before an operation, e.g. a query or an update, and after the operation. By subtracting the numbers the operation CPU cycle count is acquired, which is then divided by the CPU work frequency to get the result in seconds. Before running the RDTSC instruction, the CPU pipeline has to be cleared in order to avoid out-of-order execution, where instructions are not necessarily performed in the order they appear in the source code. Therefore, the CPUID instruction, which ensures proper CPU cycle count reads, is used before each RDTSC instruction.

In order to minimize inaccuracy due to process switching by the operating system, all of the tests are rerun a fixed number of times. After a fixed number of reruns, standard deviations and means are calculated for each of the timing results, e.g., for the average update time, and for the average range query time. If the standard deviation divided by the mean is more than 1%, the tests are run again, until it is less then 1%. At this point the fastest run is returned.

6.2 Alignment in memory

Data stored in main-memory is said to be aligned on an X byte boundary, if the start address of the data is a multiple of X bytes. An unaligned memory access, e.g. on 64 byte cache line system an access of a 64 byte data structure aligned on a 16 byte boundary, is broken up and turned into two aligned cache line fetches. This is because the memory bus can not perform unaligned cache line accesses to main-memory. The same thing happens at the lower CPU level, e.g. on a 64 bit system when accessing an 8 byte pointer aligned on a 4 byte boundary, the access is split into two aligned 8 byte accesses (the standard size of 64-bit CPU registers is 8 bytes), i.e. two 4-byte pointer parts are loaded into two different CPU registers. Therefore, unaligned access to memory represents a huge degradation of performance.

The standard c++ dynamic allocation function *malloc* typically (depending on the operating system) allocates memory where the start address is a multiple of 16 bytes, i.e. the allocated memory is aligned on a 16 byte boundary. In order to avoid unaligned accesses, a different allocation function *posix_malloc* is used. This function can allocate memory on a given boundary. Therefore we use this allocation function in our indexes when allocating data structures on a cache-line size boundary, e.g., when allocating nodes in the R-Tree or allocating buckets in the Grid.

7 Experimental Setting

In this section we present the experimental workloads used to evaluate the indexes. We furthermore describe the purpose of the individual experiments, and describe the test setup.

7.1 Test Data

We create a number of benchmark experiments, each defined by a set of workload parameters. As a point of reference, we have identified a *default* workload with settings that represent a scenario, which we consider realistic for the type of LBS application described in the Introduction. The set of default values are listed in Table 5. In each experiment, we alter a single parameter value, in order to determine the influence which each individual parameter value has on index performance. The values of the parameters are selected to reflect many different settings, including some extreme cases to determine the versatility of the indexes.

First, we briefly describe the values in Table 5. We monitor 100,000 objects in an area of 100 x 100 km. Each object is assigned with a speed of either 12.5, 25, 37.5 or 50 m/s. A reasonable index lifetime is offered by the 2,000,000 updates, which means that on average, objects are updated 20 times during one workload. The objects move between 500 hubs, which gives an object distribution with a low level of clustering. 1,000 queries are performed in total, distributed as 600 current-time (ct-RQ) and 200 predictive range queries (p-RQ), and 200 kNN queries (kNN). The range queries cover 0.5% of the monitored area and the predictive queries project object positions between 0 - 30 seconds into the future. The kNN queries locate the 100 objects that are closest to the query point. As discussed in Section 2.2, we use the point-based update policy, and set the threshold to 100 meters in the default case.

We conduct eight different experiments, and in the following subsections, the detailed goals and experimental parameters are described.

Experiment 0: Default parameter values

In this experiment, we test the performance of the indexes with the default test data to get a point of reference for the remainder of the tests.
Parameter values:
See Table 5.
Number of workloads:
1.

Experiment 1: Number of objects

This experiment is conducted to test the scalability of the indexes. The number of objects is increased while the total number of updates is kept constant.
Parameter values:
Objects = 200, 400,...,1000K.
Number of workloads:
5.

Parameter	Value
Objects	100,000
Space	100,000 x 100,000 m
$\text{Speed}_i, i = 1, .., 4$	12.5, 25, 37.5, 50 m/s
Total Updates	2,000,000
Hubs	500
Query Quantity	1,000
Query Types	ct-RQ, p-RQ, kNN; 600:200:200
Query Size	0.5%
Query Prediction offset	30 s
Number of NNs	100
Update Policy	Point-based
Threshold	100 m

Table 5: Default workload parameters used in experiments.

Experiment 2: Index lifetime

In this experiment, the effect of extending the index lifetime is examined by increasing the total number of updates while keeping the number of objects constant. This test is done to determine to which extent the indexes degrade when run longer than the default time.
Parameter values:
TotalUpdates = 4, 8,...,20M.
Number of workloads:
5.

Experiment 3: Position skew

The purpose of this experiment is to determine the performance of indexes when objects are clustered. A high level of positional skew means that the objects generated tend to be clustered in one part of the monitored area. With few hubs, the objects will tend to be clustered, and with many hubs they will tend to be more evenly distributed. If no hubs are defined the objects will be distributed uniformly.
Parameter values:
Hubs = 5,10,15,20 (Very high skew); Hubs = 50,100,150,200 (average skew); Hubs = 1000,2000,3000,4000,5000; Hubs = 0 (uniform distribution).
Number of workloads:
14.

Experiment 4: Maximum Speeds of Objects

This experiment tests the effects of varying the distribution of speeds among objects (Part 1), as well as the the effects of varying maximum speeds (Part 2). Fast objects are updated more frequently per time unit in the Point-based update policy scheme, so the update frequency increases with higher speeds.
Parameter values:
Part 1 (Distribution of speeds): All objects are assigned with either speed 12.5 m/s or 50 m/s, and fractions of objects travelling 50 m/s are: 0.1; 0.5; 0.9.
Part 2 (Equal maximum speeds for all objects): Speed = 1.67; 16.67; 33.33; 50; 166.67 m/s.
Number of workloads:
3 for Part 1; 5 for Part 2.

Experiment 5: Position accuracy threshold

This experiment is conducted in order to examine the effects of varying the distribution of thresholds among objects (Part 1), as well as the effects of varying the threshold for all objects (Part 2). The update rate depends on the threshold, so the simulation time increases when updates become infrequent.
Parameter values:
Part 1 (Distribution of thresholds): All objects are assigned either threshold 100m or 1000m, and fractions of objects with threshold 1000m are: 0.1; 0.5; 0.9.
Part 2 (Equal thresholds for all objects): Threshold = 10, 250, 1000 m.
Number of workloads:
3 for Part 1; 3 for Part 2.

Experiment 6: Query types

This experiment tests the performance of the three different query types: Current-time, predictive range queries and kNN queries. The tests are performed with the default test set, but instead of intermixing the three query types only one type of query is performed in each test. The number of queries is kept at 1,000 per workload.
Parameter values:
Query Type Ratio = (1 : 0 : 0), (0 : 1 : 0), (0 : 0 : 1).
Number of workloads:
3.

Experiment 7: Query parameters

The purpose of this test is to determine how the indexes handle queries when the query parameters are changed. In the current-time range queries the query area size is changed to see how this affects the performance (Part 1). In the predictive range query, the prediction offset is changed to determine how well the indexes handle the query expansion and subsequent filtering of results (Part 2). In the kNN queries, different k-values are tested to determine the effect of having to identify few or many Nearest Neighbours (Part 3).

Parameter values:
Part 1 (spatial extents): QueryTypes = 1 : 0 : 0, QuerySize = 0.05, 0.25, 1, 5, 10% of queried area.
Part 2 (temporal extents): QueryTypes = 0 : 1 : 0, QueryWindow = 6s, 60s, 5min, 10min, 50min.
Part 3 (object extents): QueryTypes = 0 : 0 : 1, k = 1, 10, 100, 1000.
Number of workloads: 5 for Part 1; 5 for Part 2, 4 for Part 3.

7.2 Test Setup

The tests are conducted on a machine with an Intel Pentium D 2.8 GHz CPU (64-bit Dual Core), 1 GB of RAM and a Debian 64-bit operating system. The L1 cache is 32KB (16KB for instructions and 16KB for data), L2 cache is 1MB and the cache line size is 64 bytes. The tests are run with the highest possible process priority in the operating system.

8 Results

In this section, we first discuss how the optimal index parameters were determined. Then we discuss the results of running the workloads described in the previous section. Finally, we show the average update cost composition for the Grid and the R-tree.

8.1 Determining Optimal Index Parameters

In the previous section, the different test scenarios relevant for benchmarking the indexes were discussed. In order to perform the tests and see the effect of changing different test data parameters, it is necessary to determine the best performance of the indexes in a default setting. In this way we determine the optimal default index parameters.

The indexes have different individual parameters, which were introduced in Section 5, and finding the optimal values for the indexes is performed in three phases. In the first phase the index parameters are tested with a large step size to determine a general performance trend and find an interval where the total running time is the lowest. In the second phase the test is performed with a smaller parameter step size and using the results from this test the third test is performed with the smallest parameter step size. Combining the test data, we generate a graph that shows the overall trend of the index while showing the optimal area with enough detail that we get an exact optimal default value. The different values defining the size of a *node* and a *bucket* are chosen to be a multiple of a cache line size, i.e. 64 bytes on the tested system.

The test data parameter values in the default test set are shown in Table 5. The default test workload is assumed to be a fair, average use-case scenario and therefore the total running time of a run is assumed to be a fair measure of the optimal settings for the indexes.

8.1.1 Optimal Grid Parameters

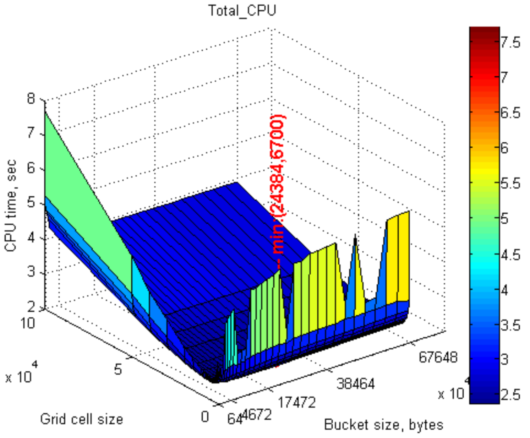


Figure 13: Determining optimal Grid parameters for total CPU time.

As shown in Figure 13 the optimal parameters for the Grid index are: a grid cell size *gcs* of 6,700, which results in a total of 225 grid cells, and a *bucket_size* of 24,384 bytes. The *CPUtime* axis represents the total CPU time taken to complete the test run., i.e. total running time. The CPU time is quite high with very low values of grid cell size, i.e. approximately from 100 to 1,000, due to the high cost of non-local updates. With the higher values of the grid cell size, i.e. approximately from 10,000 to 100,000, the trend of the total CPU time is going up due to the poor performance of the queries, because performing a query becomes more like a linear scan of the data. With low values of the *bucket_size*, i.e. approximately from 64 to 1024, the total performance suffers from frequent memory allocations and deallocations when updating and inefficient queries. The effect of the *bucket_size* is discussed in more detail in Section 8.1.3.

8.1.2 Optimal R-Tree Parameters

In Figure 14 the *CPUtime* axis represents the total CPU time taken to complete the test run., i.e. total running time. Therefore Figure 14 shows the total running time of the R-Tree when running the default test set with different values of *fanout* and *minchildren*. The *minchildren* does not affect index performance noticably. The *minchildren* at the optimal point is 25%, but the difference between the *minchildren* with the longest and shortest runtime is less than 5%. When testing the index with different parameters we have found that the nodes of the

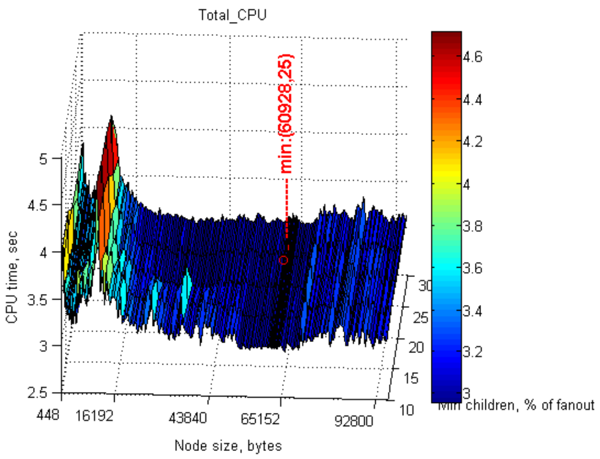


Figure 14: Determining optimal R-tree parameters for total CPU time.

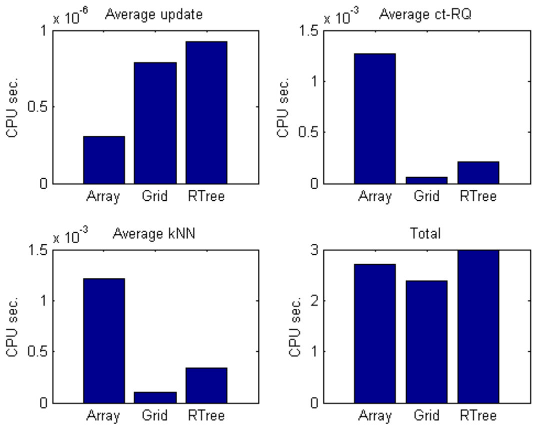


Figure 15: Test on the default data set.

R-Tree are approximately 55%-60% full on average, ranging over the different parameter settings. The running time of the default test begins high with a very small *node_size*, whereafter it drops near 4Kb which is enough to contain approximately 160 children in every node. With nodes filled to 60% capacity on average, the height of the R-Tree is 3. It peaks around 10.5Kb (approx. 430 children maximum) where the cost of scanning individual nodes and the height of the tree combine to make updates expensive. This is because the tree height is reduced to 2, but the sequential scanning of nodes during non-local updates and queries makes it inefficient. After the peak it flattens out because the height of the tree is 2 and the children of the root become fewer as their individual capacity becomes larger. If all the nodes were completely full, the tree structure would have flattened out at 7.5Kb (approx. 317 children), but because of the average node fullness the flattening out is shifted. The lowest running time is at 61Kb (approx. 2,500 children) where the root node contains 55 children.

8.1.3 Conclusion on Optimal Index Parameters

As observed in the results in Sections 8.1.1 and 8.1.2 the total times tend to reach the minimum once reaching a certain bucket size or fanout in the Grid and R-tree, respectively. At some point, the index substructures become big enough to contain the elements that are assigned to them. After this point, there is no reason to increase the size of a node or a bucket for the default workload because the performance gain is minimal and the space consumption becomes unnecessarily big. In the Grid this occurs when there is no need for more than 1 bucket for every cell, and in the R-tree it occurs when the fanout results in only a root node and 1 additional level of nodes that contain the objects. As explained by Drepper [4], the reason why bigger substructures are more efficient is that the hardware prefetching ensures that the latency of memory accessing is hidden by pipelining from memory to cache. Another factor is that sequential memory accesses can be accomplished with very high sustained data rates because less time is spent on precharging and preparing accessing of new rows (setting the $\overline{RAS}$ [2]). This shows that memory locality is an extremely important issue when working in main-memory.

8.2 Performance of the Indexing Structures

In this section, the results of performing the tests described in the previous section are shown. Each set of test results is displayed and analysed. In Figure 16, the legend for all of the following plot figures is displayed. In the following figures 'Average update' refers to the average time in seconds that it takes to process one update,'Average ct-RQ' refers to the average time per current-time range query, 'Average p-RQ' refers to the average time per predictive range query, 'Average kNN' refers to the average kNN query time, and 'Total' refers to the total runtime of the experiment.

8.2.1 Experiment 0: Default data set

In this experiment the indexes are tested on the default workload. As illustrated in Figure 15, the average update in the Array is about 2 and 3 times faster than in the Grid and the R-Tree, respectively. The Grid updates about 20% faster than the R-Tree. The clear advantage of using more sophisticated data structures is observed from the performance of queries, e.g. in range queries, the Array is approx. 20 and 6 times slower than the Grid and the R-Tree, respectively. The total runtime is about the same with a small variation in all indexes.

[2] Row Address Selection ($\overline{RAS}$) defines which row in main-memory is selected for reading/writing [4]

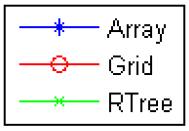


Figure 16: Legend of figures.

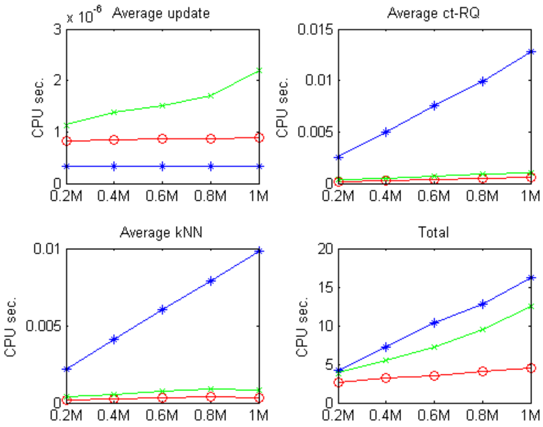


Figure 17: Number of objects.

8.2.2 Experiment 1: Number of objects

The graphs in Figure 17 show that the indexed number of objects does not influence the average cost of performing an update in the Array and the Grid. In the R-Tree the average update cost is increasing with the number of indexed objects, because of the increased height of the tree and the increased number of expensive node splitting and merging operations. The Array shows a significant decrease in query performance because the queries are performed with a linear scan through the Array and the performance therefore depends on the number of objects that need to be scanned. The total runtime graph shows that the performance of the Grid degrades less than the other indexes when increasing the number of indexed objects.

8.2.3 Experiment 2: Index lifetime

In this experiment, indexes are tested on the increasing number of total updates. In Figure 18 the average cost of operations shows that none of the indexes degrade over time, i.e. over the number of handled updates. The total time is increasing because the number of updates is increasing, whereas the average cost of an update stays about the same.

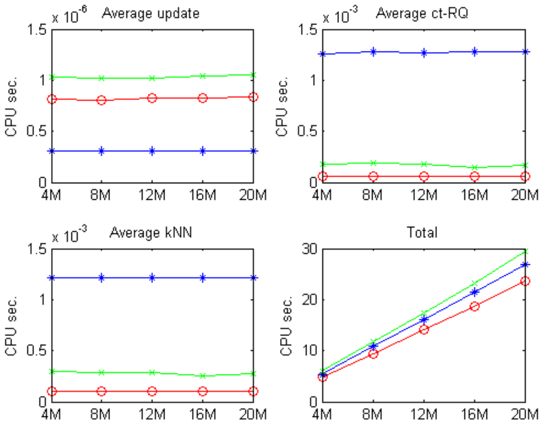


Figure 18: Index life time.

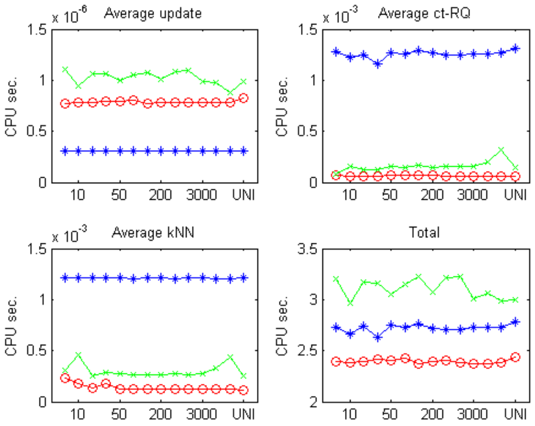


Figure 19: Position skew.

8.2.4 Experiment 3: Position skew

In this experiment, the indexes are tested on the varying object distribution, from a very clustered object distribution (a low number of hubs used when generating the data) to a uniform object distribution. Results of the position skew test are shown in Figure 19. As for the average update performance there is no significant change in any of the indexes. The same holds for the queries in the Array, whereas the cost of an average range query in the R-Tree is increasing with the increasing number of hubs. In the Grid index the average kNN query performance is mostly affected by a very clustered object distribution, i.e. by the low number of hubs. This is because there is no spatial ordering within a list of a bucket, i.e., with a clustered object distribution, the kNN query is treated more like a linear scan than an efficient best-first on-demand traversal of cells.

8.2.5 Experiment 4: Maximum Speeds of Objects

In part 1 of this experiment, various distributions of speeds among objects are tested. The result graphs of this part of the experiment are displayed in Figure 20. There is no significant change in the indexes when changing the distribution of fast and slow objects.

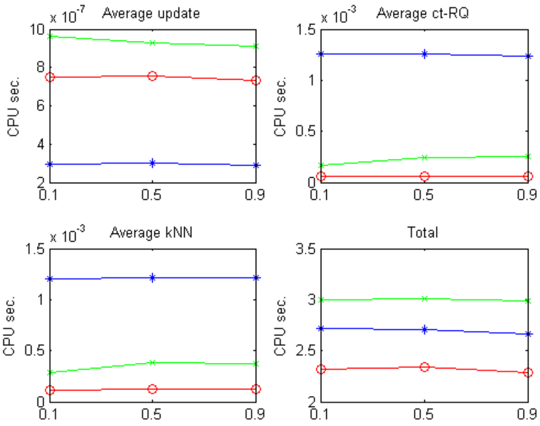


Figure 20: Maximum speed of objects, part 1

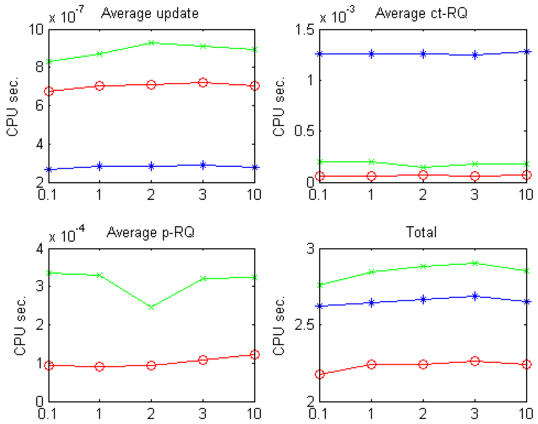


Figure 21: Maximum speed of objects, part 2

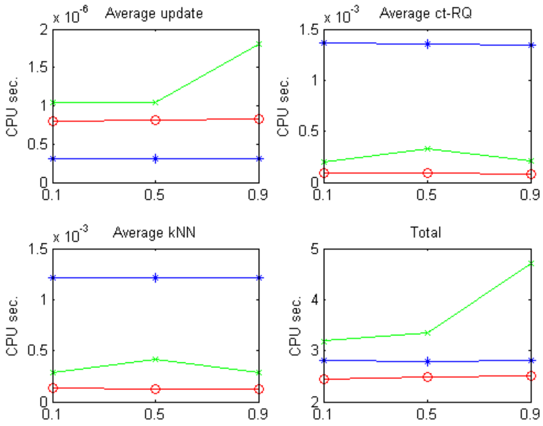


Figure 22: Position accuracy threshold, part 1

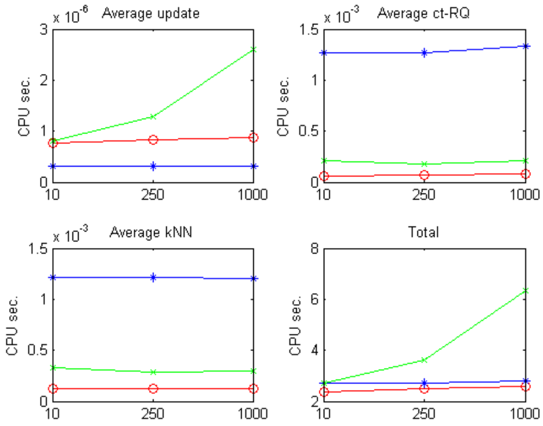


Figure 23: Position accuracy threshold, part 2

In part 2, the effect of varying maximum speeds is tested. Results of this part are displayed in Figure 21. There is no significant change in performance when the maximum speed of objects is increased. The reason for this is that we are using the point-based update policy, which means that no matter how fast the objects are moving they will trigger updates at the same spatial positions. The increased speed only affects the indexes in predictive range queries where the expansion of the query area is bigger with higher speeds. The conclusion is that there is a direct correlation between increasing the speed of objects and the number of updates in a given time interval.

8.2.6 Experiment 5: Position accuracy threshold

In part 1 of this experiment, various distributions of thresholds among objects are tested and results are displayed in Figure 22. The average update cost in the Array is not affected by the distribution of objects with different thresholds. In the Grid the cost of an update increases slightly because a larger number of objects change grid cells on updates. In the R-Tree the average cost of an update increases significantly for a higher ratio of objects that use a larger position accuracy threshold. This is because a larger position accuracy threshold makes updates less local. When updates are less local, bottom-up updating becomes more expensive because the upwards traversal will, on average, go higher up the tree before finding a suitable subtree to insert the object in. The queries are not affected in the Array and the Grid, whereas in the R-Tree, queries cost slightly more with an even distribution of objects with different threshold values.

In part 2, effects of varying the threshold for all objects is tested and the results are presented in Figure 23. The cost of an average update is not affected in the Array, slightly affected in the Grid and significantly increases in the R-Tree when increasing the threshold value. The increased thresholds

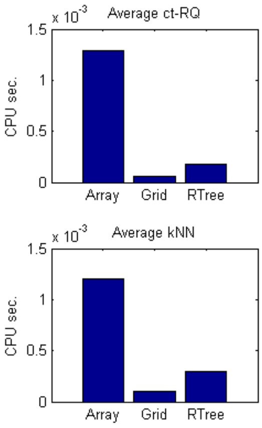


Figure 24: Query Types

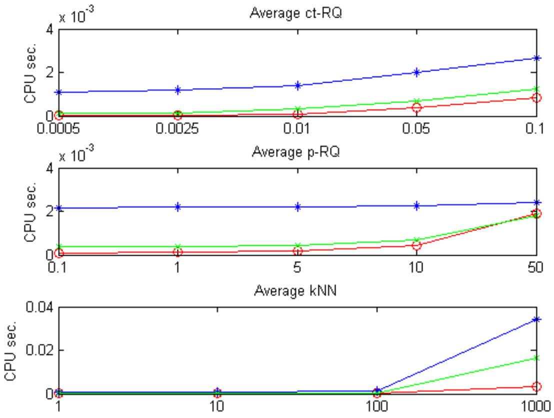


Figure 25: Query Parameters

affect the Grid due to the increasing number of non-local updates. This also the cause of ineffciency in the R-tree. The average cost of performing a range query slightly increases in all of the indexes due to the expansion of the query by the position accuracy threshold value. The kNN queries are not affected by the changing position accuracy threshold.

8.2.7 Experiment 6: Query types

In Figure 24, performance graphs for the different types of queries are displayed. The range query is about 3 times faster in the Grid than in the R-Tree, whereas in the R-Tree it is about 6 times faster than the Array. The cost of a predictive range query is higher than the cost of a simple range query in all of the indexes, althogh the relative cost difference between the indexes is about the same. The kNN query is also about 3 times faster in the Grid than in the R-Tree, although the R-Tree is only about 4 times faster than the Array.

8.2.8 Experiment 7: Query parameters

The effects of changing the query parameters are displayed in Figure 25. Increasing the current-time range query coverage increases the running time in all of the indexes, because a bigger result list is constructed. There is an additional cost increase in the Grid and the R-Tree because more grid cells and nodes have to be checked in range queries with a bigger coverage area.

The array is not affected much by the increasing prediction offset in the predictive range query, because the size of the result list does not change significantly due to the filtering of objects. In the Grid and the R-Tree, a higher value of the prediction offset means a bigger expansion of the query, which results in checking more grid cells and nodes, hence the cost of a predictive range query increases. The Grid becomes less efficient than the R-Tree with a prediction offset higher than approximately 40 minutes.

No sigificant change is observed when changing the k value of a kNN query from 1 to 100 in all the indexes. However, when changing from 100 to 1000 the Array is affected the most, because the result list constructed in the Array is a linked list ordered on the distance to the query point in a descending order. Complexity of inserting into that list is $O(n)$. The same holds for the Grid and the R-Tree result list, although due to the specifics of the algorithms, the elements inserted into the list tend to be closer to the start of the list. The performance of the kNN queries in the Grid and the R-Tree is affected more by a higher number of cells or nodes visited when increasing k from 100 to 1000. Of all the indexes, the kNN algorithm used in the Grid is least affected by the increasing number of nearest neighbours to be found.

8.3 Average update cost composition

In this section the cost of an average update is analyzed by breaking it down into smaller sub-operations. The measurements are done on the default data set with the optimal index parameters. The average cost of the sub-operations is measured by using the technique described in Section 6.1. Afterwards the average sub-operation cost is multiplied by the number of its occurrences during updating and divided by the total number of updates. Finally, the number is divided by the average cost of an update to get the relative cost of the sub-operation. This way the composition of an average update cost is constructed. The cost of initial object inserts into the indexes is not taken into account. Therefore the total number of updates is $2,000,000$. In the following subsections the compositions of an average update cost in the Grid and the R-Tree are presented and discussed.

8.3.1 Average update in the Grid

As illustrated in Figure 26, the average update cost in the Grid is mainly composed of 6 sub-operations.

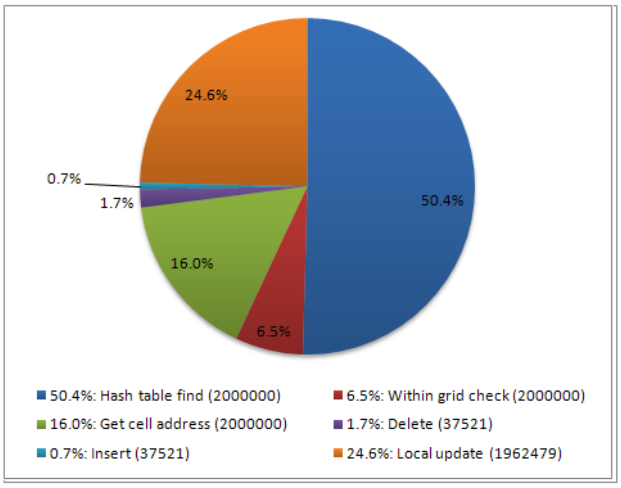


Figure 26: Composition of the average update cost in the Grid.

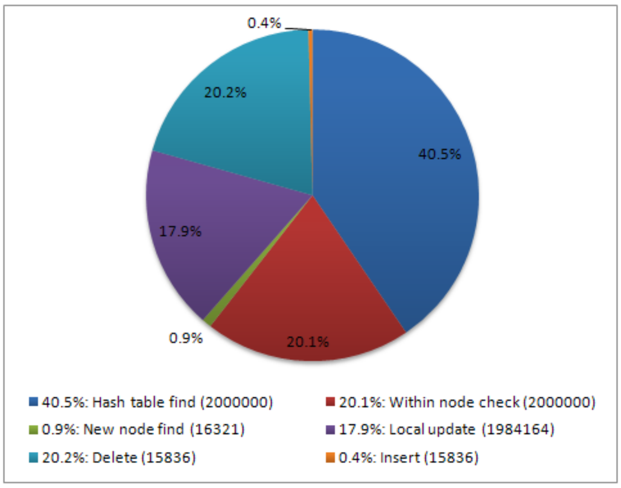


Figure 27: Composition of the average update cost in the R-Tree.

The number of sub-operation occurrences during updating is given in the parenthesis. The most expensive one is the *Hash table find* sub-operation, which is 50.4% of the cost of an average update. This can be improved by using a different hash table implementation as discussed in Appendix A. 24.6% time of an average update is spent on the *Local update*, which only involves the updating of object data in the index. 16% of the cost is spent on calculating the cell address that corresponds to the new update. Checking if the update is within the boundaries of the Grid index takes 6.5% of the cost. Finally, the lowest cost of the two sub-operations, *delete* and *insert*, is due to the fact that only 37521 grid cell changes occur, which is only about 1.8% of total updates.

8.3.2 Average update in the R-Tree

In Figure 27 the composition of the cost of the average update in the R-Tree is displayed. The cost is mainly composed of 6 sub-operations. The number of sub-operation occurrences during updating is given in the parenthesis. As in the Grid the most expensive sub-operation is *Hash table find*, which takes up 40.5% of the cost. As in the Grid this can be improved by using a different hash table implementation as discussed in Appendix A. Checking if the object remains within its current node is 20.1% of the cost. The *delete* sub-operation occurs only in 15836 updates, but its relative cost to the average update is quite high - 20.2%. This is because, the *delete* involves expensive linear scans of siblings when recalculating MBRs, because the MBRs might have to be contracted. This takes up 97% on average of the time spent on the *delete* sub-operation. This does not occur in the *insert*, because the MBRs only need to be expanded, which does not involve linear scans. The *Local update* takes up about 17.9% of the average update, whereas locating the new node for an object is 0.9% of the cost. Object insertion into a new node takes up 0.4% of the cost. The difference between the occurrence of *New node find* and *insert/delete* is $16321 - 15836 = 485$. This is because in 485 updates the new optimal node found for the updating object is the same as its old one, but the node MBR needs to be epxanded.

8.4 Summary

As observed in Section 8.1 the indexes can be fine tuned by determining the optimal parameters for a particualar setting of hardware and data. An interesting observation is that the optimal *bucket_size* and $fanout$ values determined are hundreds of times larger than the cache line size, which suggests that the optimal structures in a memory setting should be flat to preserve locality as much as possible.

The performance results from Section 8.2 show that the Array is superior in updating, but slow at performing queries. Regarding the performance of the advanced index structures, the Grid is generally superior to R-tree, although not significantly. R-tree queries are slow with low values of $fanout$, because of the MBR overlaps. They are still slower than in the Grid with a high value of $fanout$, due to the longer linear scans than in the Grid. Since the R-tree is more versatile, both proposed designs are good solutions for indexing moving objects in a main-memory setting. In some settings, the total CPU cost of the Array is similar to that of the advanced indexes. However, since end-users of LBSs value the performance of queries more than the performance of updates, the Array is a less desirable solution.

As seen in Section 8.3 both indexes spend the most time on hash table look-ups when updating. Therefore, using a better hash table implementation would improve the update performance significantly in both of the indexes.

9 Conclusion

We present the problem of creating update-efficient main-memory structures that index the positions of dynamic spatio-temporal objects. Compared to traditional databases, LBS applications that offer current-time and near-future predictive queries are characterized by requiring much more efficient update handling while maintaining fast query times, posing a huge challenge to the design of index structures. However, LBSs typically do not index past object positions, which makes persistence less of an issue. Thus, main-memory indexes are well-suited for the task, and for the past decade, a number of such indexes have therefore been proposed.

Existing main-memory indexes are hard to compare, as they are presented in different settings. In fact, it is unclear, whether a space partitioning or a data partitioning approach is superior for the purpose of indexing moving objects in main-memory.

We use the COST benchmark [7] in a main-memory setting, as it is specifically designed for evaluating the performance of these indexes. Using an update policy, we ensure that updates occur only when it is required in order to meet guaranteed position accuracies, within a specified threshold. We define a variety of experiments and generate corresponding workloads, which simulate a variety of situations that could occur in the real world, along with a number of extreme cases that evaluate the robustness of the indexes.

We propose two main-memory variants of spatio-temporal indexing structures: the Grid and the R-tree, which represent a space partitioning and a data partitioning approach to update-efficient dynamic object indexing, respectively. Additionally, we offer an implementation of a non-spatial Array index, which acts as a point of reference for the results of the other indexes.

We evaluate how the indexes handle a variety of current-time and predictive range queries, as well as kNN queries, to determine the CPU processing time, when running different workloads. From the results we conclude that the Grid is generally faster than the R-tree in both updating and querying. Non-local updates are handled noticeably slower in the R-tree, which is anticipated since the tree structure has to be maintained. This means that the Grid is superior when it comes to the total running time.

The Array provides the fastest updates and the slowest queries. For some experiments, the total running time is similar to the Grid and the R-tree. However, we find that the high query times make it ill-suited for LBSs, where end-users value fast query responses higher than fast updates.

The *bucket_size* in the Grid and the $fanout$ with *min_children* in R-Tree are the parameters that can be used to fine tune the indexes for a particular setting. The fact that the grid cell size *gcs* and the total monitored area have to be defined before using the Grid, makes the Grid a less attractive solution than the R-Tree, which is more versatile since it does not need any information about the monitored area.

In the future, we would like to evaluate the performance of other existing index structures such as the *TPR-tree* in a main-memory setting, to determine whether the fewer amount of updates compensate for the added operation costs of each inividual update, when indexing object trajetories. Another interesting direction is to explore the effects of introducing compression techniques like in the *CSB+-tree* and the *CR-tree*. In addition, there are further optimizations for the proposed indexing structures to explore, e.g. adding space-filling curves in the Grid and optimizing the MBR recalculations in the delete operation of the R-Tree.

10 Acknowledgement

We would like to thank our project supervisor Simonas Šaltenis for proposing the problem, for useful discussions, and for providing guidance and valuable feedback on our work.

We also want to thank our fellow students for many useful discussions about the problem.

References

[1] Arthur C. Ammann, Maria Hanrahan, and Ravi Krishnamurthy. Design of a memory resident dbms. In *COMPCON*, pages 54–58, 1985.

[2] Shekhar Borkar. Getting gigascale chips: Challenges and opportunities in continuing moore's law. *Queue*, 1(7):26–33, 2003.

[3] Alminas Civilis, Christian. S. Jensen, Jovita Nenortaite, and Stardas Pakalnis. Efficient tracking of moving objects with precision guarantees.

[4] Ulrich Drepper. What every programmer should know about memory.

[5] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[6] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient b+-tree based indexing of moving objects. In *vldb'2004: Proceedings of the Thirtieth international conference on Very large data bases*, pages 768–779. VLDB Endowment, 2004.

[7] Christian S. Jensen, Dalia Tiesyte, and Nerius Tradisauskas. The cost benchmark-comparison

and evaluation of spatio-temporal indexes. In *DASFAA*, pages 125–140, 2006.

[8] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing multidimensional index trees for main memory access. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 139–150, New York, NY, USA, 2001. ACM Press.

[9] J. Kuan and P. Lewis. Fast k nearest neighbour search for r-tree family, 1997.

[10] Mong-Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *VLDB*, pages 608–619, 2003.

[11] Jussi Myllymaki and James H. Kaufman. Mobile data management (part 3) - dynamark: Benchmarking dynamic spatial indexing for location-based services. *IEEE Distributed Systems Online*, 4(12), 2003.

[12] Jun Rao and Kenneth A. Ross. Making b+-trees cache conscious in main memory. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 475–486, New York, NY, USA, 2000. ACM.

[13] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. pages 71–79, 1995.

[14] Shashi Shekhar, Sanjay Chawla, Siva Ravada, Andrew Fetterer, Xuan Liu, and Chang tien Lu. Spatial databases-accomplishments and research needs. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):45–55, 1999.

[15] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 331–342, New York, NY, USA, 2000. ACM Press.

[16] Ouri Wolfson, A. Prasad Sistla, Sam Chamberlain, and Yelena Yesha. Updating and querying databases that track mobile units. *Distrib. Parallel Databases*, 7(3):257–387, 1999.

[17] Wei Wu and Kian-Lee Tan. isee: Efficient continuous k-nearest-neighbor monitoring over moving objects. In *SSDBM '07: Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, page 36, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. Lugrid: Update-tolerant grid-based indexing for moving objects. *mdm*, 0:13, 2006.

[19] Man Lung Yiu, Yufei Tao, and Nikos Mamoulis. The bdual-tree: indexing moving objects by space filling curves in the dual space. *VLDB J.*, accepted for publication, 2008.

Appendices

A Secondary Index

The choice of a hash table implementation has a significant influence on the index update performance. One choice is the hash table implementation *hash_map* from the Standard C++ Library. This implementation is used in the indexes of this paper. A possible alternative is the google implementation of *hash_map*, which has two variants, a *dense_hash_map*, which is more time efficient, and a *sparse_hash_map*, which is more space efficient.

B Call stack

A call stack is a dynamic stack data structure which stores information about the active subroutines of an application. In many programming languages, including c++, the call stack contains a limited amount of memory, usually determined at the start of the program. The size of the call stack depends on many factors, including the programming language, machine architecture, multi-threading, and amount of available memory. When too much memory is used on the call stack, e.g. in an excessively deep or infinite recursion, the stack is said to overflow, which typically results in the crash of the application.

C Framework

For the implementation of the indexes we created a framework consisting of common data structures and virtual functions that need to be implemented in all the indexes. This was done to ensure comparability between the indexes because both the input and the output from different indexes would in the same format. The following is a list of common data structures:

Point: A data structure used to contain two dimensional values. It is a template structure that consists of an x-value and a y-value of a given data type. Examples of usage is as a pair of integer values describing coordinates or a pair of float values describing an object velocity vector.

ObjectData: A data structure used to contain object information. This data structure is used for updates and query results. It consists of a long value identifying the object, an integer Point coordinate and a float Point velocity vector.

ObjectDataList: A data structure that extends the ObjectData structure with a pointer, making it possible to create linked lists of ObjectData.

Linked: A data structure used to create linked lists for kNN queries. It is not used by the framework, but it has been used in all implementations of the kNN query function and it was added in the framework to avoid multiple definitions. It is a template structure containing a pointer to the next element in the linked list, an unsigned long long to contain the distance between two points and a variable data type that can be used to store the actual data of the list.

Rectangle: A data structure which is used in range queries to define a query rectangle. It consists of two integer Point's which describe the maximum and minimum points of the queried area.

The functions defined in the index framework do not contain any implementation details, but they clearly define the formats for input and output from the indexes. In this way using an index which has been built on the framework requires no knowledge of the actual implementation of a given index. The following is a list of the functions defined in the framework:

insertObject: inserts a new object in the index without returning anything. It uses ObjectData to define the information of the object.

deleteObject: deletes an object from the index. For all of the implemented indexes, the only information used is the object ID, but this is because every index is either using a hash table to locate the object or the index is directly addressable using the object ID as a key.

updateObject: is used when updating an object in the index. It contains the updated object data.

rangeQuery: is used to perform current time range queries. It defines the query area using the Rectangle data structure and returns a pointer to a linked list of the objects that were within the query.

predictiveRangeQuery: works in the same way as the rangeQuery function, but uses a prediction offset to calculate the expansion of the query range.

kNNQuery: finds the k objects nearest to the query point p and returns the results in a linked list of ObjectData.

D Test Automation

In order to automate the testing as much as possible, config files are used. A library $libConfig{+}{+}$ is used in order to achieve that goal. Within a config file,

```
Test1{
  verbose = "yes";
  runs = "10";
  Grid{
    gcs{"100";"200";}
    bucket_size{"64";"512";}
  }
  Data{
    query_verification = "no";
    datafiles_q{
      kNN_k{"100";"200";}
      files{"data1.sql";"data2.sql";}
    }
  }
}
Test2{
...
}
```

Table 6: An example of a config file.

a number of tests can be defined, as shown in Table 6.

Within a test specification, parameter *verbose* denotes how much output to the screen is shown and parameter *runs* defines how many reruns of each test are ran, of which average values of timing measurements are taken. In the test section *data* the input data is defined. Parameter *query_verification* is used to turn on and off the query verification explained in Section E. The subsection $datafiles\_q$ defines the input data files. Within $datafiles\_q$ a number of kNN query k values kNN_k and data files $files$ can be defined, which are then iteratively used for testing. In future work parsing of different input data can be implemented. In the config file it could just be a different subsection of section *Data*, e.g. *sql_data*.

Within a test specification section *Grid* denotes the index that should be used for testing. The section names can also be $R - tree$ and *Array*, which respectively denote the use of an R-tree or the simple array. Within a test all three indexes can be specified. Each of the index specifications, contain index parameters, e.g. *gcs* and *bucket_size* for the *Grid* index section. A number of index parameter values can be defined. Tests are run, by taking each data input, i.e. a data file, loading it into memory, then iteratively measuring timings of each of the indexes with all combinations of the parameters defined. As mentioned before, average timing results are taken by rerunning each test a fixed number of times (*runs*).

E Query verification

Original index descriptions are usually high level specifications of the structures and algorithms. A lot of implementation details are left out, e.g. the grid kNN query description [17] is for an infinite grid, whereas in a real implementation a grid has boundaries and is not infinite. Therefore a particular programmer has to be very careful when implementing an index, because even if the index appears to work, it might not actually do what it is supposed to do.

One way to verify the correctness of a index is by veryfing the query results. The correct results of a query are acquired by doing a linear scan of the objects, i.e. querying the simple array. Querying the array is straightforward and simple, therefore it is assumed that the simple array query implementation is good and the query results are correct. This assumption is validated, when other indexes return the same results, because it is quite improbable that all of the indexes return the exactly the same wrong results.

All three types of queries, i.e. range, predictive range and kNN queries, are implemented in the array index, therefore all three types of queries can be verified. The result of a query in a particular index is compared to the result of the query in the array index, if any mismatch is found, both query results and the mismatch is output to the screen for debugging purposes.

After each major modification of the implementation and before doing the final experiments for this paper queries are reverified. The query verification is turned off for timing measurments used to compare the indexes. This query verification, adds the possibilty to easier correctly implement any other indexes in the framework.