Indexing Moving Objects using Short-Lived Throwaway Indexes

Jens Dittrich[1], Lukas Blunschi[2], and Marcos Antonio Vaz Salles[2]

[1] Saarland University jens.dittrich@cs.uni-saarland.de
[2] ETH Zurich lukas.blunschi@inf.ethz.ch
[3] Cornell University vmarcos@cs.cornell.edu

Abstract. With the exponential growth of moving objects data to the Gigabyte range, it has become critical to develop effective techniques for indexing, updating, and querying these massive data sets. To meet the high update rate as well as low query response time requirements of moving object applications, this paper takes a novel approach in moving object indexing. In our approach we do *not* require a sophisticated index structure that needs to be adjusted for each incoming update. Rather we construct conceptually simple *short-lived throwaway indexes* which we only keep for a very short period of time (sub-seconds) in main memory. As a consequence, the resulting technique MOVIES supports at the same time high query rates *and* high update rates and trades this for query result staleness. Moreover, MOVIES is the first main memory method supporting time-parameterized predictive queries. To support this feature we present two algorithms: non-predictive MOVIES and predictive MOVIES. We obtain the surprising result that a predictive indexing approach — considered state-of-the-art in an external-memory scenario — does not scale well in a main memory environment. In fact our results show that MOVIES outperforms state-of-the-art moving object indexes like a main-memory adapted B^x-tree by orders of magnitude w.r.t. update rates and query rates. Finally, our experimental evaluation uses a workload unmatched by any previous work. We index the complete road network of Germany consisting of 40,000,000 road segments and 38,000,000 nodes. We scale our workload up to 100,000,000 moving objects, 58,000,000 updates per second and 10,000 queries per second which is unmatched by any previous work.

1 Introduction

Indexing support for moving objects is a crucial requirement in domains such as car tracking [18], airplane surveillance [45], mobile phone tracking [1], emergency services [10], social networking [24], and gaming engines [49]. In these applications an *update* may be a car/airplane/phone sending a message on its new position. A *query* may be a range query, a nearest-neighbor query, or a time-parametrized range query asking for predicted positions of moving objects at a future time t_q. Queries are issued either by car/air traffic control or by users themselves. All of the above applications face a principal problem: how to support efficient query processing under high update rates.

Traditionally, index creation has been considered an extremely costly process. For that reason, research on moving object indexes has been centered around creating sophisticated index structures. These indexes are created once, kept, and then modified

according to incoming updates. This has led to a plethora of complex index structure proposals in the past. However, with the rise of large main memories and fast multi-core CPUs this "natural law" of keeping a moving object index can be questioned.

We present a novel main-memory method termed MOVIES (MOVing objects Indexing using frEquent Snapshots) that supports time-parameterized (predictive) queries and is at the same time space-, query-, update-, and multi-CPU-efficient. At its core our method *MOVIES* resembles the approach taken by a cinematographer: as it is impossible to capture continuously moving data with any camera in one image, a cinematographer has to take a series of *still* images at a given *frame rate*. As long as the frame rate exceeds the inertia of the human eye (i.e., at least 24 frames per second), an illusion of continuous movement is created. We follow exactly the same approach: we try to provide as many still *index images* of the data as possible. For a very short period of time we use that index to answer incoming queries. After that, we throw that index away. As long as the *index build rate* is high, an illusion of a continuously up-to-date index will be created. We will show that — surprisingly — index creation can be a matter of subseconds even for datasets comprising hundreds of thousands of moving objects. For instance, we will demonstrate that index creation for 1 million moving objects (a common data set size used in recent moving objects studies, see Section 6.4) takes as little as 0.16 seconds on a single computing core allowing for six index rebuilds per second. The price we have to pay for these features is slightly out-of-date (stale) query results, i.e., even though queries are executed immediately in our approach, query results may not consider the most recent updates. However, we will show that even for massive data sets this query result staleness may be reduced to (sub-)seconds. This meets by far the demands of real applications. For instance, state-of-the-art flight control in Europe currently works with a staleness of 5 seconds [39].

1.1 Contributions

In summary, this paper makes the following contributions:

1. We provide a novel approach termed MOVIES (MOVing objects Indexing using frEquent Snapshots) to effectively index moving objects. As described above MOVIES resembles the approach taken by a cinematographer by creating a series of different indexes each second. Like that we provide at all times a read-optimized index not suffering from update handling.
2. MOVIES is the first main-memory moving object index to support time-parameterized queries. This allows users to pose predictive queries asking for predictive results at a future time t_q. Previous main memory approaches such as [51] did not support this type of query. We will present two different MOVIES variants to support these type of queries: *Non-Predictive Indexing MOVIES (NPI)* and *Predictive Indexing MOVIES (PI)*. We will show that MOVIES NPI performs better than MOVIES PI for high update rates. This is a surprising result as predictive indexing approaches are considered state-of-the-art for external memory methods.
3. We present techniques to make update handling efficient. As we collect incoming updates in a buffer, the cost for collecting the updates is very small. We will show that different buffer organizations have different impact on the overall performance of MOVIES. Therefore, we will propose two more variants of our algorithm: *Logged* MOVIES and *Aggregated* MOVIES.
4. We provide a thorough experimental study of MOVIES using standard hardware and realistic data sets unmatched by any previous work. Our experiments show that MOVIES scales

well up to 25 million moving objects on a single machine. We show that MOVIES provides excellent query response times while at the same time being able to process huge amounts of updates. In addition, we show that MOVIES outperforms state-of-the-art indexing methods like the B^x-tree by orders of magnitude w.r.t. the number of queries and updates being handled — even though the latter methods have been adapted to work effectively in main memory. Finally, we evaluate a distributed implementation of MOVIES indexing 100 million moving objects on a small cluster of shared-nothing machines. Note that the data sets used in our experiments are 10 times larger than in the biggest study available [23] and at least 100 times larger than in all other studies.

This paper is structured as follows. The following section presents preliminaries. Sections 3&4 present MOVIES. Section 5 presents our experimental evaluation. Section 6 discusses related work and its relationship to MOVIES.

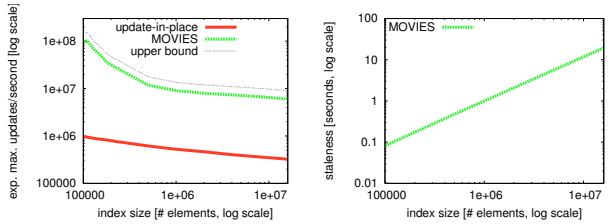
2 Preliminaries

2.1 Problem Statement

We consider a data set of N moving objects in a two-dimensional space of data of a domain $|X| \times |Y|$ where $|X|$ (resp. $|Y|$) represents the number of different positions in the horizontal (resp. vertical) dimension. Extending our technique to more dimensions is straightforward. Similarly to [17], we assume a discrete space of $2^{16} \times 2^{16} = 2^{32}$ different positions. Each moving object is identified by a unique key termed an *OID*. Each moving object emits updates on its current location (x,y) and its speed vector $\overrightarrow{sv}$ by sending an $(x,y,\overrightarrow{sv},OID)$-tuple to central indexing server(s). Like in [17], we assume that objects travel at a maximum speed S_{max} and are guaranteed to send updates at least every $t_{\Delta max}$ seconds. We assume that indexes are queried with two-dimensional predictive range queries $Q(r,t_q)$ specifying a range $r = [x_{low};x_{high}] \times [y_{low};y_{high}]$ and a query time t_q. Note that other query types such as predictive k-nearest-neighbor may easily be derived from predictive range queries (see e.g. [17]).

2.2 Formal Argument

In this section we provide a formal argument to illustrate the core benefit of our approach. We do not strive to provide a full-blown cost model but rather focus on the key aspects. For realistic moving objects scenarios the amount of updates will be in the tens of millions per second. We will develop a method that does not trade query performance for update performance as done in several existing methods. Consider a simple index structure organizing a sorted mapping $spatial position \mapsto OID$ (binary range search on a B+tree or any cache-optimized tree). We assume that the spatial position is linearized using a linearization function (see Section 3.3). The cost for both querying and updating in-place are of the order $O(log\,N)$ where N is the number of entries. An update in a positional index consists of deleting the old entry and creating a new entry. Thus in the worst case we need two logarithmic traversals. We derive a cost formula $C_{\text{update-in-place}} = 2 \cdot c_1 \cdot \log_2 N$ where c_1 is a hardware-dependent constant. Similarly, the initial cost for bulkloading for an index is of the order $O(N\,log\,N)$, which translates to a cost formula



(a) expected maximum update rate (the gain)

(b) expected staleness (the price)

Fig. 1. expected performance of MOVIES versus update-in-place

$C_{\text{bulkloading}} = c_2 \cdot N \log_2 N$ where c_2 is a hardware-dependent constant. Now let's assume that instead of performing updates in-place we collect W updates in a separate structure with $O(1)$, i.e., $C_{\text{array update}} = c_3$. We will peridodically rebuild a new index from that structure. The cost for this is $C_{\text{collect and rebuild}} = W \cdot C_{\text{array update}} + C_{\text{bulkloading}}$. When will this be cheaper than update-in-place? We obtain $W \cdot c_3 + c_2 \cdot N \log_2 N \leq W \cdot 2c_1 \cdot \log_2 N$ $\Rightarrow c_2 \cdot N \log_2 N \leq W \cdot (2c_1 \log_2 N - c_3) \Rightarrow W \geq (c_2 \cdot N \log_2 N)/(2c_1 \log_2 N - c_3)$. Now, we may estimate upper bounds for the constants assuming a single core and the index to be limited to 16 million elements as $c_1 = 73.6ns$, $c_2 = 8.9ns$, and $c_3 = 112.5ns$. Thus we receive $W \geq (8.9 \cdot N \log_2 N)/(2 \cdot 73.6 \cdot \log_2 N - 112.5)$. For an index of $N = 1,000,000$, the collect and rebuild approach will already be cheaper when the number of updates reaches $W = 62,872$. Also note that the query processing costs in both approaches are *exactly the same*. We just argued on how to improve update cost without touching query cost. On the contrary, the collect and rebuild approach could even be improved to build read-optimized indexes. That would additionally improve the query response time over an update-in-place approach. Now let's examine the maximum number of updates supported by the different methods. How many updates can we expect to support in a collect and rebuild approach? We may rebuild the index every $T_{\text{frame time}} > C_{\text{collect and rebuild}}$ seconds. This can be rewritten to $C_{\text{collect and rebuild}}/T_{\text{frame time}} \leq 1 \Rightarrow W \cdot C_{\text{array update}} + C_{\text{bulkloading}} \leq T_{\text{frame time}} \Rightarrow W \leq (T_{\text{frame time}} - C_{\text{bulkloading}})/(C_{\text{array update}})$. The maximum number of updates processed per second can then be computed as $U^{max} = W/T_{\text{frame time}}$ which is limited by the *upper bound* $1/C_{\text{array update}}$. Assume we allow for a $T_{\text{frame time}}$ of $3C_{\text{bulkloading}}$, then we obtain the function displayed in Figure 1(a). The figure shows that we may expect to *gain* an order of magnitude over update-in-place. The *price* we pay for that is query result staleness which is limited by $2T_{\text{frame time}}$. Figure 1(b) shows that even for an index of 1,000,000 elements staleness will remain below a second even when using only a single computing core.

3 MOVIES

This section presents the MOVIES indexing algorithm (MOVing objects Indexing using frEquent Snapshots). As stated in the Introduction, our method resembles the approach taken by a cinematographer: we try to create as many still index images as possible. This generates the illusion of a continuously up-to-date index.

3.1 Algorithmic Walkthrough

The MOVIES algorithm is based on *index frames*. Each index frame is active during a short time interval called the *frame time* $T_i = [t_i;t_{i+1})$ where i denotes the ID of the frame and t_i denotes the moment in time when frame i started. During each index frame, e.g., time interval T_{45} for Frame 45 in Figure 2, we use a read-only index, I_{44}, to answer all incoming queries. We also keep an update buffer U_{45} collecting all updates arriving during T_{45}. In addition, we build a new read-only index I_{45} based on the updates collected in update buffer U_{44} during T_{44} (see arrow ⟵). Depending on whether the update buffers contain updates for all OIDs, this index build has to consider information available in index I_{44} (see arrow ⟵ - - -). As soon as the new index I_{45} is built, we start a new frame, e.g., Frame 46. In this frame we use the newly built read-only index I_{45} from Frame 45 to answer all incoming queries. We keep an update buffer U_{46} to collect incoming updates. In addition, we build a new read-only index I_{46} based on the updates collected in U_{45} during T_{45}. Again, depending on whether the update buffers contain updates for all OIDs this index build has to consider information available in index I_{45}. As soon as the index is built, we start a new frame, e.g., Frame 47 (not shown) which is similar to Frame 45.

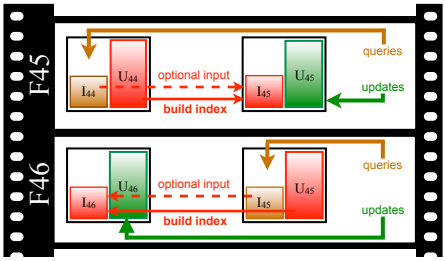


Fig. 2. two index frames of MOVIES core algorithm

3.2 Comparison to Differential Files

The idea of collecting updates in a separate space and applying them in a batch was first used in the context of relational databases more than 30 years ago [38]. The idea of that paper was to collect changes in a separate *differential file* and merge that file regularly with the existing external memory index. Since then differential files were extended in multiple ways [29,16,28] and became state-of-the-art for *read-mostly* environments like data warehouses (DWH) [46] as well as desktop [25], enterprise [46], and web search [11] engines.

In contrast to all of these approaches MOVIES differs as follows:

1. For a moving objects application the query result staleness of a file-based method as followed in other applications [46,25,11] would be unacceptable. For moving object indexing we require query result staleness to be below a few seconds (e.g., for an aircraft surveillance scenario it should be below 5 seconds [39]). Therefore we have to optimize our algorithm for keeping staleness low. This can only be achieved by keeping the data entirely in main-memory.
2. In our scenario moving objects are guaranteed to send an update at least every $t_{\Delta max}$ seconds. This was used in similar studies, e.g. [17]. Therefore for certain situations, e.g. $t_{\Delta max} < T_i$ we may completely ignore the information available in the old index: we simply need to build an index image from the update buffer. Therefore, in contrast to differential file-based approaches [29,16,28], in MOVIES there is no need to perform a costly *merge* with the previous index. An index merge will only be used as a fallback.
3. Finally, in order to support time-parameterized queries we need to introduce timestamp-consistent predictive indexes (MOVIES PI). Thus, instead of indexing data as-is as in differential file-based approaches, we will predict data to a future point in time into the index.

3.3 MOVIES Core Algorithm

MOVIES core algorithm is displayed in Figure 3.

It takes as input a stream of updates $StreamU$, a stream of queries $StreamQ$, and a $bootstrapTime$ interval. The algorithm starts by creating a new update buffer U_0 (Line 1). Then the stream of updates $StreamU$ is routed to that buffer (Line 2). An empty index I_0 is created in Line 3.

Then the algorithm waits for a certain amount of time specified by a $bootstrapTime$. During that time, however, incoming updates are collected in update buffer U_0. Lines 5–15 show the indexing loop used to create the sequence of index frames. This loop will be repeated until a global flag `should_terminate` is set to true (Line 5). The loop keeps a counter $currentFrameID$ for the current frame ID. Inside the loop an index frame starts by creating a new update buffer $U_{currentFrameID}$ (Line 6). The stream of updates is routed to the new update buffer (Line 7).

```
   Input: Stream of updates StreamU
   Stream of queries StreamQ
   TimeInterval bootstrapTime
 1 U_0.create()
 2 StreamU.setDestination(U_0)
 3 I_0.create()
 4 wait(bootstrapTime);
 5 for (Integer currentFrameID = 1; ¬should_terminate) do
 6     U_currentFrameID.create()
 7     StreamU.setDestination(U_currentFrameID)
 8     StreamQ.setDestination(I_currentFrameID−1)
 9     if currentFrameID ≥ 2 then
10         I_currentFrameID−2.destroy()
11     end
12     if ("may ignore old index") then
13         I_currentFrameID ← buildIndex(U_currentFrameID−1)
14     else
15         I_currentFrameID ← buildIndex(U_currentFrameID−1, I_currentFrameID−1)
16     end
17     U_currentFrameID−1.destroy()
18     currentFrameID = currentFrameID + 1
19 end
```

Fig. 3. MOVIES Core Algorithm

The stream of queries is routed to the index created in the previous iteration (Line 8). For the first loop iteration this index will be the empty index I_0. In Line 9 we check whether the $currentFrameID$ is two or higher. If that is the case, we destroy the index built in index frame $currentFrameID-2$ (Line 10). After that we check whether we may ignore the data available in the old index (Line 12), e.g., this may happen if all elements in the old index became outdated by elements in the update buffer. If that check succeeds, we simply call the `buildIndex` operation on the update buffer (Line 13) otherwise we create a new index $I_{currentFrameID}$ also using information from the old index (Line 15). After that we destroy the update buffer filled in the previous frame (Line 17). Finally, we increase the $currentFrameID$ counter by one and continue looping (Line 18).

Organization of Update Buffers This section describes the organization of update buffers. As we want to handle high update rates, we have to make sure that the update buffers do not exceed the available main memory. We solve this as follows. For high update rates the update buffers may contain several updates for the same *OID*, i.e., the update buffer may be considerably larger than the original index. As the aggregate of updates to an *OID* is sufficient for query processing (e.g., the most current position of a moving object), it makes sense to implement update buffers U_i by *aggregation buffers* $\hat{U}_i$, organized using *OID*s as keys. For each *OID*, $\hat{U}_i$ only keeps a `MIN` aggregate, i.e., the last update received for this moving object. In our approach updates are written to aggregation buffers $\hat{U}_i$ immediately when they arrive. We implemented aggregation buffers using arrays of size N where the slot at position i stores the aggregate of object $i = OID$. Note that other implementations are possible, e.g., any hash table. This ensures

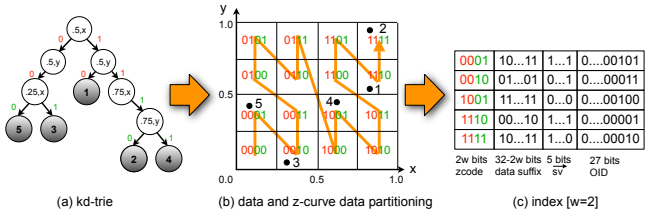


Fig. 4. kd-trie mapped to a pointer-free linearization

constant insert time for updates. We refer to this variant as *Aggregated MOVIES*. The algorithm based on FIFO update buffers is termed *Logged MOVIES*.

Organization of Read-Only Indexes As read-only index we use a state-of-the-art spatial indexing method. We focus on a technique that is at the same time simple and efficient. Therefore we have chosen linearized kd-tries [47,32][4]. This index was used in many papers in different variants (e.g., [36,31,9,17]) and was shown to outperform competing approaches. The core idea of a linearized kd-trie is to *simulate* a pointer-based index structure. This is achieved by assigning each node of a virtual kd-trie a unique identifier termed a *locational code*. Locational codes are based on a space-filling curve like the z-curve [47,32] (see Figure 4(b)) or the Hilbert curve [14]. These recursive space-filling curves enumerate a multi-dimensional space (i.e., the nodes of the kd-trie, see Figure 4(a)) with a one-dimensional curve (see Figure 4(b)). For each object that needs to be managed by an index, it then suffices to compute its locational code, i.e., the virtual node it belongs to in the kd-trie. This calculation is independent of the locational codes of other objects. Therefore, at no point in time it is required to actually create a pointer-based kd-trie. As the locational codes are one-dimensional, they are ordered to provide efficient query processing. The resulting codes plus the data are then stored in a sorted index (see Figure 4(c) using $w = 2$ bits per dimension). Note that locational codes may be inlined with the data to avoid extra storage cost. Both point and range queries are efficiently supported. The latter are crucial for our scenario as several other types of queries such as nearest neighbor queries may be based on range queries (see e.g. [17]). Moreover, kd-tries are not limited to two dimensions but also work well for high dimensional spaces. Also recall that in contrast to grid-based indexes (e.g., as used in [51]) which would need to store an exponential number $(1/grid\_length)^d$ of pointers to inclusion lists, an approach based on locational codes will for all d only require N indexing slots. Thus, our method can easily be extended to higher dimensional data.

4 MOVIES Query Processing

4.1 Time-Parameterized Query Processing

A time-parameterized query $Q(t_q)$ asks for object positions in the range $([x_{low};x_{high}] \times [y_{low};y_{high}])$ at a time t_q. In order to answer these queries, we must transport objects to their predicted positions at time t_q. This can be done at indexing time, which we term *Predictive MOVIES (PI MOVIES)* or at querying time, which we term *Non-Predictive*

[4] Following the terminology of Donald Knuth [20] a *trie* partitions the data space whereas a *tree* partitions the data.

MOVIES (NPI MOVIES). Both strategies work for both Logged and Aggregated MOVIES resulting in a total of four different combinations.

Predictive MOVIES (PI) Indexing Strategy. For each index build we index all data w.r.t. a *single* point in time $t_{\text{index}} > t_u$. We term t_{index} the *index time*. Thus, for every incoming update u we index the moving object at a predicted position $(x,y) + \overrightarrow{sv}(t_{\text{index}} - t_u)$. Here, we may avoid any extra storage space by translating objects immediately when an update arrives. However, for each incoming update we have to compute the predicted position — which may be costly. After that, the timestamp for the update may be dropped. If during fallback an object is encountered that has not received an update (Line 22 of the `buildIndex` algorithm), that object is simply translated to a new position using the new index time.

Query Strategy. As t_q may be either larger or smaller than t_{index} we have to consider three cases:

1. $t_q < t_{\text{index}}$: the objects have to be translated to an earlier time,
2. $t_q > t_{\text{index}}$: objects have to be translated to a later time (see also [17]),
3. $t_q = t_{\text{index}}$: objects do not have to be translated.

For cases (1)&(2) we rewrite $Q(t_q)$ to consider the maximum distance $\varepsilon := S_{max} \cdot |t_q - t_{\text{index}}|$ an object may have travelled relative to the index time. Every $Q(t_q)$ is rewritten to $Q(t_q)' := [x_{low} - \varepsilon; x_{high} + \varepsilon] \times [y_{low} - \varepsilon; y_{high} + \varepsilon]$. $Q(t_q)'$ is then postfiltered w.r.t. t_q and their respective speed vectors $\overrightarrow{sv}$ as obtained from the index.

Choice of Index Time. Let $\bar{Q} = \{Q_1(t_q^1), .., Q_k(t_q^k)\}$ be the set of queries arriving during a single indexing frame of MOVIES. To minimize query enlargements and therefore the performance penalty for large range queries, we wish to minimize the sum of query window areas added due to enlargement, or alternatively the term $\sum_{i=1}^{k} |t_q^i - t_{\text{index}}|^2$. Assume, for simplicity, that all queries ask for a fixed time into the future, i.e., $t_q = t_{\text{now}} + \Delta t$. After being rebuilt, an index receives queries during one frame time $T_{frametime}$. To produce balanced query enlargements, the index time should be $\Delta t + T_{\text{frame time}}/2$ ahead of the time the index is ready for querying. In order to achieve that, we must set the time of the update buffer that will be used to build a new index appropriately. An update buffer is used to collect updates two frames before being used to build a new index (see Figure 3). Therefore, the index time to be used for an update buffer collecting updates in the next frame should be set to $t_{\text{index}} = t_{\text{now}} + 2T_{\text{frame time}} + (\Delta t + T_{\text{frame time}}/2) = t_{\text{now}} + 2.5T_{\text{frame time}} + \Delta t$.

Non-Predictive MOVIES (NPI) Indexing Strategy. For each index build we index each index entry w.r.t its timestamp t_u. In order to do this, we need to keep for each update its corresponding timestamp. Thus, we require slightly more storage space, but do not have to compute predicted positions at indexing time.

Query Strategy. We use query enlargement like with Predictive MOVIES. However, as objects are indexed with different last update times, we must perform query enlargement with respect to the time of the oldest update considered in the index, i.e., $\varepsilon := S_{max} \times |t_q - t_{\text{min}}|$. As calculating t_{min} by scanning the timestamp array has prohibitive cost, we provide a bound on t_{min}. When an update arrives, it takes at most two times the frame time $T_{\text{frame time}}$ for it to appear in the index used for querying (see Figure 3). As an update for each object must arrive within $t_{\Delta\text{max}}$, then $t_{\text{min}} \geq t_{\text{now}} - (t_{\Delta\text{max}} + 2T_{\text{frame time}})$.

5 Experiments

This section presents a thorough experimental analysis of MOVIES. We explore various aspects of our approach and compare it with state-of-the-art approaches. The goals of our experiments are:

1. Determine the maximum supported update rate of MOVIES when scaling the index size (Section 5.3).
2. Determine query throughput of MOVIES when scaling the update rate (Section 5.4).
3. Determine the performance of MOVIES when implemented on a cluster of shared-nothing machines (Section 5.5).

5.1 Setup

All experiments were performed on servers having each two 2.4 GHz Dual Core AMD Opteron 280 processors, i.e., four cores in total, and 6 GB of main memory. We used two separate servers M1 and M2 to generate updates and queries. These updates and queries were sent over the network and received by servers termed *processing nodes* (PN1–PN4) that did the actual indexing. M1 and M2 were each connected to the switch by one network cable and in addition with one network cable each directly to PN3 (PN4 respectively). All network links supported 1 Gbit/s.

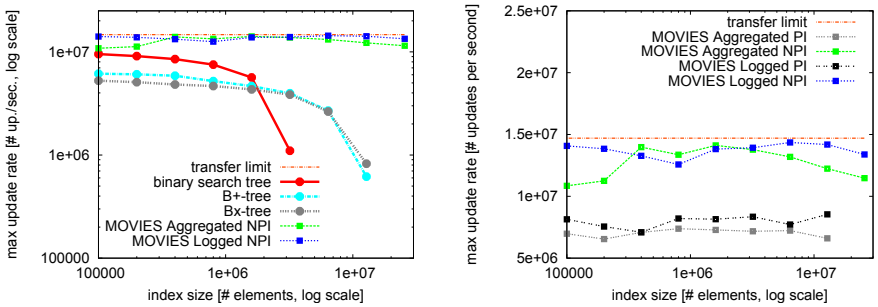
For the single instance experiment we used one processing node PN1. For the parallelization experiment we used up to four processing nodes PN1–PN4. All code used for the experiments was implemented in Java 5. In our implementation, we avoided complex object types whenever possible and used primitive Java types. To make maximal use of the four cores provided by each machine, we implemented a multi-threaded variant of MOVIES. For the experiments in Sections 5.3 to 5.5 evaluating MOVIES, we waited until at least 8 re-indexing phases were completed. Then we measured at least 10 re-indexing phases and report the average.

parameter	setting
index size N	100,000 ... **6,400,000** ... 100,000,000
update rate V	0 /s ... 58,000,000 /s
query rate	0 /s ... **1,000 /s** ... 10,000 /s
query window size qw	**1 km**×**1 km** .. 10 km×10 km
# road network segments	39,509,805
# road network nodes	37,967,339
data region	640 km×863 km
index granularity	26.3 m×26.3 m
S_{max}	60 m/s
$t_{\Delta max}$	$\leq N/V$

Fig. 5. Settings

5.2 Data and Queries

Our experiments were inspired by the scenario *'index all cars in Germany'* which comprises 58M cars [22]. We obtained a commercial data set containing the complete road network of Germany [44]. This data set consists of 38 million nodes and 40 million road segments. The geography of Germany covers 640 km x 863 km. We assumed cars to travel at a maximum speed of $S_{max} = 60$m/s$= 216$km/h. As in similar studies [26,27,34] we initially used the moving object generator of [5]. However, it turned out that that generator does not scale for the massive workloads considered in this paper. In particular, large number of nodes and road segments are not possible. Therefore we developed our own generator based on the ideas of [5]. This means that we used the same moving object placement strategy *network-based approach (NB)*. It places cars using the same skewed distribution as the roads and nodes themselves. We then moved cars on



(a) max update rate: comparison of MOVIES, binary search tree, B^+-tree, and B^x-tree

(b) max update rate: comparison of four different variants of MOVIES

Fig. 7. Scalability in index size for high update rates [query rate=1,000/s], single processing node

the network by assigning each car a constant random direction which it would try to follow on the network. This avoids the overheads of doing Dijkstra-computations for each car. This generates similar traces as in [5], but at much lower cost. Our trace generator is open source and available from sourceforge at *http://moto.sourceforge.net/*. If not mentioned otherwise, a data set of 6.4 million moving objects was used on MOVIES experiments with a single processing node. For scalability experiments on a single processing node, we used up to 25 million moving objects (raw size = 200MB).

For the parallelization experiment we used up to 100 million moving objects (raw size = 800MB). As outlined in the Introduction, we assumed all data and indexes for all methods to fit into main memory. We evaluate time-parameterized predictive range queries. Note again that other query types such as time-parameterized predictive k-nearest-neighbors may be inferred from predictive range queries (see [17]). We used a query window size corresponding to the size of a small town center like Oldenburg which has an extension of roughly 1,000 m $\times$ 1,000 m. Bigger query windows did not substantially change our results and therefore we do not show those results. Query centers were chosen using the NB strategy [5] thus creating a skewed distribution on queries. If not mentioned otherwise, we set a query rate of 1,000 queries per second and a query time $t_q = t_{\text{now}} + 1.5t_{\Delta max}$. Figure 5 summarizes the settings.

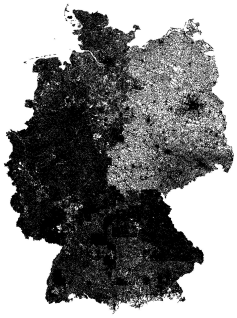


Fig. 6. Road network of Germany

5.3 Scalability in Index Size

The goal of this experiment is to understand the maximum update rates supported by MOVIES when scaling the size of the data set. We compare MOVIES to baseline index structures, including binary search trees and B^+-trees. Furthermore, we compare MOVIES against a state-of-the-art moving object index: the B^x-tree [17]. As all tree-based methods are hard to parallelize without considerably sacrificing performance

(locking), we parallelized all tree-based methods to obtain lock-free methods as follows: we partitioned the data by *OID* into four disjoint partitions, and used a separate tree and thread to index each partition as suggested in [40]. Thus, the tree-based methods could make maximal use of the four cores available on a server. All methods evaluated in this and the following experiments could make use of the same amount of main memory which was set to 5.5 GB. In particular, all tree-based methods resided completely in main memory. Therefore, at no point any disk-I/O was performed. We tuned the node size of the trees to obtain the best possible performance in a separate experiment. Only the best tree-based methods are displayed.

Figure 7(a) shows the results of a scalability experiment where the index size is varied up to 25.6 million moving objects. We kept a fixed query rate of 1,000 time-parameterized queries/s and display the maximum update rate supported by each indexing method. The results show that both variants of MOVIES outperform all other methods. Figure 7(a) shows that all tree-based methods degrade sharply with growing index sizes. The binary search tree was not able to scale beyond 3.2 M objects as then it could not meet the query rate anymore. For the B$^+$-tree, we experimented with several values of k and k^*. However, the best B$^+$-tree we could devise ($k = k^* = 16$) was not able to scale beyond 12.8M moving objects. For 12.8M the B$^+$-tree could only handle 0.6M updates/s. Similarly, for the B$^\mathbf{x}$-tree we performed a separate experiment varying the number of phases n and display only the best version of the B$^\mathbf{x}$-tree we could devise ($k = 16$ and $n = 2$). Interestingly, the B$^\mathbf{x}$-tree was also not able to scale beyond 12.8M moving objects. The B$^\mathbf{x}$-tree even performed slightly worse than the best B$^+$-tree except for N=12.8M. This is due to the fact that the B$^\mathbf{x}$-tree incurs overhead compared to the B$^+$-tree as it has to compute predictions for each incoming update at indexing time.

In the experiment, in contrast to all other methods, MOVIES Logged with Non-Predictive Indexing (NPI) shows update rates of around 14 million updates/s for index sizes up to 25.6M objects. This value is close to the network limit of 14.7 million updates/s. MOVIES Aggregated NPI shows in average a slighlty smaller update rate ranging from 11M to 14M updates/s. However, MOVIES Aggregated NPI performs still better than all tree-based methods. For an index of size of 12.8 M objects the improvement of the best MOVIES variant over the best B$^+$-tree and B$^\mathbf{x}$-tree is factor 15. For an index size of 25.6 M only MOVIES was able to index the data meeting the query rate. Interestingly, in this experiment the binary search tree performs even better for small indexes than the B$^\mathbf{x}$-tree. This is due to the high cost for computing predictions for each incoming update. This is also evidenced when we compare the four different variants of MOVIES using different indexing methods. Figure 7(b) displays the update throughput for MOVIES when using different indexing schemes, i.e., either Aggregated or Logged

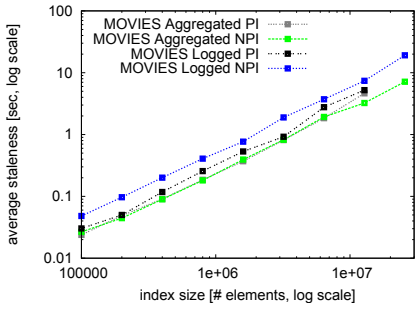


Fig. 8. Average staleness when scaling index size: comparison of four different variants of MOVIES [query rate=1,000/s]

MOVIES (Section 3.3), and either non-predictive (NPI) or predictive indexing (PI) (Section 4.1). The figure shows that Logged MOVIES NPI has the best performance. In contrast, Logged MOVIES PI achieves only half of the throughput. This is due to the fact that for predictive indexing each update has to be translated to a new position. This is CPU-intensive and also explains why the $B^{\mathbf{x}}$-tree performs worse than a standard B^{+}-tree: at extreme update rates, the computational cost of predictions offsets the gain obtained by smaller query window enlargements.

Figure 8 shows the average staleness observed in the scaling experiment. The results show that the staleness grows for larger index sizes. That is expected as an important component of frame time is the time to sort the data in a new index. For MOVIES Logged NPI the staleness grows up to 19 sec, for MOVIES Aggregated NPI it increases up to 7 sec. If the staleness has to be reduced, this can be achieved by scaling out on multiple processing nodes. This is explored in Section 5.5.

5.4 Scalability in Update Rate

The goal of this experiment is to understand the maximum query rate supported by MOVIES when scaling the update rate. We keep the index size constant at 6.4M objects and vary the update rate. Figure 9 shows the result. The figure shows that the binary search tree is only able to sustain a very low query rate. For update rates above 2.1M updates/s this method is not able to execute any more queries and thus fails to scale beyond this point. Similarly, we observe that for update rates between 0.1M to 1M the best B^{+}-tree is able to execute between 3,000 and 2,000 queries/s, respectively. For higher update rates, however, the B^{+}-tree degrades sharply: for an update rate of 4M updates/s, the best B^{+}-tree is only able to execute a small amount of queries and thus fails to scale beyond this point. The $B^{\mathbf{x}}$-tree has better query performance than the B^{+}-tree for update rates up to about 3M updates/s, but also fails to scale beyond 4M updates/s. The MOVIES variants show an interesting behavior: The predictive variants outperform the non-predictive variants in terms of query performance up to an update rate of 4M updates/s, exhibiting query rates around 3,000 queries/s. Above 4M updates/s, however, the query performance of the non-predictive variants sharply increases up to 9,200 queries/s. This behavior can be explained by analyzing the trade-off between indexing predictions and performing query window enlargements. For modest update rates, non-predictive methods must perform bigger query enlargements to compensate for the relatively large value of $t_{\Delta\max}$. These enlarged queries impose significant computational overhead. Predictive methods, on the other hand, aggressively reduce query window enlargements by computing predictions for each update applied to the index. At high update rates, however, we observe the opposite effect: predictive methods pay a high computational cost for predicting every update applied to the index. The gain in query window enlargements is not enough to offset these costs, because as $t_{\Delta\max}$ is relatively low, the enlargements performed by non-predictive methods are also relatively small. In addition, non-predictive methods have lower computational cost for collecting updates. Another effect may be observed for the non-predictive variants: at an update rate of about 10M updates/s, the query rate drops to around 6,000 queries/s. This slight drop in the query rate may be explained by the fact that at very high update rates, the cost

to collect updates starts to become significant, draining CPU resources from both query processing and index rebuilding.

The staleness for Logged MOVIES NPI stayed constant around 3 sec up to 7M updates/s. For higher updates rates it increased linearly up to 7.2 sec. For Aggregated MOVIES NPI the staleness was constant around 2.5 sec. The predictive variants could not be scaled beyond 8M updates/s and their average staleness stayed between 2 to 3 sec. The relatively high staleness for low update rates can be explained as follows: If during one index rebuild MOVIES receives only few updates, then MOVIES has to retrieve the old data for many objects from the old index. This leads to many random accesses to the old index and therefore hurts rebuild performance. The time needed to lookup old data goes down as the update rate increases and reaches zero around 5M updates/s. Even though this effect would lead to decreasing staleness, the staleness stays about constant, because processing the updates becomes more expensive. In summary, this experiment shows that the MOVIES variants scale well for high update rates. Of all methods, only MOVIES was able to scale up to 14M updates/s. Note again that all methods completely resided in main memory.

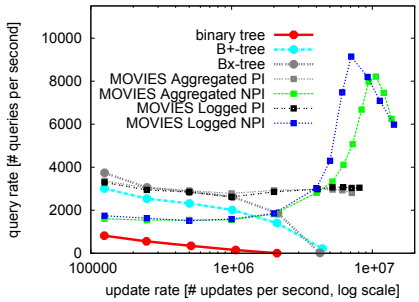
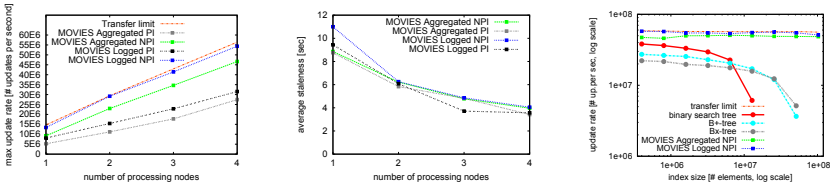


Fig. 9. Scalability in update rate: Comparison of MOVIES with binary search tree, B^+-tree and B^x-tree for high update rates [index size=6.4E6]

5.5 Shared-Nothing Scale-Out

The goal of this experiment is to examine how MOVIES scales when increasing the number of processing nodes. In order to adapt the different methods to a shared-nothing landscape, we horizontally hash-partitioned the data by OID. We keep the index size constant at 25.8M and vary the number of processing nodes PN from one to four. As our experiments with a single processing node have shown, the transfer limit imposed by the network is a serious bottleneck. Therefore, we required a special network setup as described in Section 5.1. With that setup we could transfer up to 58M updates/s to four processing nodes while still being able to distribute queries. Figure 10(a) shows the results. The NPI MOVIES variant Aggregated (resp. Logged) scales up to 47M (resp. 54M) updates/s. Figure 10(b) displays an experiment where we keep the index size constant at 25.8M and keep the maximum update rate at 5M updates/s, which is supported by the worst MOVIES method. We display the average staleness. The figure shows that staleness goes down almost linearly if we increase the number of processing nodes. For four processing nodes staleness goes below 3 seconds for all four variants of MOVIES. In summary, this experiment shows that MOVIES scales linearly w.r.t. the maximum number of updates and linearly w.r.t. to the average query result staleness.

In another experiment we used all four processing nodes for indexing. Figure 10(c) shows the results. Similarly to the single instance experiment MOVIES outperforms all other methods. All tree-based methods, including the B^x-tree degrade sharply for growing index sizes. The tree-based methods fail to scale beyond an index of size 51M,



(a) max update rate: effects of scaling number of processing nodes [index size = 25.8E6]

(b) average staleness: effects of scaling number of processing nodes [index size = 25.8E6]

(c) Scalability in index size on four shared-nothing servers: Comparison of MOVIES with binary search tree, B^+-tree, and $B^{\mathbf{x}}$-tree

Fig. 10. Shared-nothing performance [query rate = 1,000/s]

i.e., 12.8M moving objects per processing node. In contrast, MOVIES scales up to 102M moving objects. Furthermore, for index sizes up to 51M, Logged MOVIES sustains an update rate close to the network limit of 58M updates/s. For 51M moving objects the improvement of MOVIES over the best B^+-tree is factor 15; the improvement over the best $B^{\mathbf{x}}$-tree is factor 11. The average staleness of all the MOVIES variants is the same as shown for the single instance experiment, but the index size is four times larger. See Figure 8. For example, the staleness of MOVIES Logged NPI is 21 seconds for an index with 102M elements.

6 Related Work

Considerable work has been done in the area of moving objects. The existing methods can be classified into two groups: methods with or without time-parameterized (TP) queries. General design issues for moving object indexes can be found in [30].

6.1 Methods With TP Queries

External Memory. Many approaches are centered around extending external memory structures like the B^+-tree, R-tree[13], or R^*-tree[3]. All of these methods assume that data would not fit into main memory. Examples include the TR-tree and TB-tree [35], the TPR-tree [48], the TPR*-tree [42], the STP-tree [41] and the R^{PPF}-tree [34]. The most relevant work to our work is the B^x-tree [17] as, conceptually, it has some similarities to the MOVIES indexing strategy. The core idea of the B^x-tree is to map three-dimensional data (two spatial and one temporal dimension) to a one-dimensional space. This is done by using a recursive space-filling curve and mapping data to a B^+-tree very similarly to [32]. However, in contrast to the latter approach, the B^x-tree also partitions data into phases corresponding to future time intervals. For each phase it uses a separate subtree to index moving objects and predicted positions. As a consequence, prediction queries are supported. As the B^x-tree is based on a B^+-tree, it is very easy to integrate it into existing DBMSs. The B^x-tree was shown to outperform competing methods such as the TPR-tree [48]. However, in contrast to MOVIES the B^x-tree does not rebuild the

index based on updates buffers but rather follows an update strategy similar to update-in-place. Also the partitioning into phases used by the B^x-tree leads to relatively high query cost (as observed in our experiments) which is avoided by MOVIES. Other methods index moving objects by transforming them to a higher dimensional space. This includes STRIPES [33] and [21] which transform d-dimensional space to $2d$-dimensional Hough-X space [15]. The recently proposed B^{dual}-tree [50] uses the same idea; however it maps the Hough-X space back to a one-dimensional space using a Hilbert curve. [43] presents a study on dual methods concluding that if query efficiency is required (as required in this paper), dual methods are not competitive. Interestingly, in the concluding remarks of [43] it is suggested that it could be beneficial to rather reconstruct a non-dual method periodically. Exactly this approach is followed by MOVIES.

Main Memory. The approach of [7] partitions data into sets of active objects that stay in a main memory buffer and inactive objects that reside on external memory. Therefore that work is more of a buffering scheme for moving object indexing. It is orthogonal to the techniques presented here and can be applied on top of any moving objects index.

6.2 Methods Without TP Queries

Main Memory. Relevant to our work are methods that use main memory for monitoring queries. The method of [19] uses a fix-sized grid where the grid-size is chosen w.r.t. the average query window sizes. Each grid cell maintains pointers to two lists with query results. Query results are periodically reevaluated and query results are delivered with a time delay Δt. [51] extends [19] to k-NN queries. [26] improves [51] to only update grid-cells that are affected by an incoming update. However, none of the former methods provides any support for time-parameterized queries. Also [19,51,26] do neither provide any means how to scale for cases when the main memory is exhausted nor provide any parallelization scheme. In contrast MOVIES provides solutions for all of these issues. [27] focusses on k-NN in road networks where the distance among objects is not the euclidean distance but rather the length of the shortest path on the network. Therefore the latter method will not work for objects not following roads, e.g., planes, ships, people's phones. In contrast, MOVIES supports all of these scenarios.

6.3 Extensions for Efficient Updates

External Memory. Frequent update handling in R-trees was treated in [23,4]. A general survey on how to optimize B-trees for high update rates was recently presented by Graefe [12]. Several of these optimizations may be traced back to Lars Arge's buffer tree [2]. Graefe also mentions differential files [38] as an effective means to trade query performance for update performance. However, [12] does not mention that one could trade query result staleness and keep *both* queries and updates efficient as in MOVIES.

Main Memory. Batching updates in a similar way to Lars Arge's buffer tree [2] was also considered for main memory optimized trees such as [52,8] however trading query for update performance. In contrast, MOVIES does not trade query performance for update performance. Other cache-efficient trees are the CSS-tree [37] and the FPB+-tree [6]. An interesting challenge would be to extend both the B^x-tree and MOVIES

to include these optimizations. However, as pointed out in Section 2.2, the query processing performance is not affected by MOVIES. Therefore, the general trade-off of update-in-place versus collect and rebuild as used by MOVIES will remain unchanged. Rather, as MOVIES may build read-only indexes at each index frame, MOVIES could even improve overall query performance by building read-only cache-aware indexes.

6.4 Experimental Studies

Moving object scenarios comprise a large number of objects and a large number of updates. As mentioned above, the number of cars in Germany is about 58,000,000 [22]. Assume every car sends an update on its position every 2 seconds, then this boils down to 29,000,000 updates per second. If we were to index not only cars but also planes, people's cellular phones, etc., we would face even higher data and update volumes. In this work we are interested in supporting these large scale scenarios. Therefore we are considering data sets of up to 100,000,000 moving objects. This is 10 times larger than in the biggest study available [23] and by at least two orders of magnitude larger than in all other studies, e.g., [26,27,17,19,51,7]. We think it is important to scale to such large data sets in order to understand the limits of the different methods.

7 Conclusions

This paper has proposed a novel approach to time-parameterized moving object indexing of massive data sets under very high update rates. Our approach is based on frequently building short-lived throwaway indexes. This keeps at the same time query throughput high, query response time low, and update performance high. The price we have to pay is slightly out-of-date (stale) query results, which is acceptable in several applications including aircraft control [39]. We have shown that this price can be reduced to be as small as a few seconds even for very large data sets of up to 100,000,000 moving objects. Our experiments have demonstrated the feasibility of our approach even for massive realistic data sets. We have presented results of an experimental study using the entire road network of Germany: a network size unmatched by any previous work. In our study we scale up to 100,000,000 moving objects and 58,000,000 updates per second. MOVIES shows order of magnitude improvements over state-of-the-art approaches like the B^x-tree, as well as several baseline methods w.r.t. supported update rates and query rates. One general conclusion is that the popular pattern of *keeping and modifying an index* should be dropped for moving object scenarios. Another surprising conclusion of our study is that the idea of indexing predictions for time-parameterized queries as done by some external memory indexes does only work well in main memory for low update rates. In terms of future work we plan to examine the trade-off of scalability and stalenesss in more detail. Another research direction would be to extend our approach to consider cache-aware B^+-trees, e.g. [37]. However, as shown by our formal analysis, the general trade-off of update-in-place versus collect-and-rebuild would even be improved in favor of MOVIES.

References

1. I. Anderson et al. Shakra: Tracking and Sharing Daily Activity Levels with Unaugmented Mobile Phones. *Mobile Networks and Applications*, 12(2–3), 2007.
2. L. Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms (Extended Abstract). In *WADS '95*, 1995.
3. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990.
4. L. Biveinis, S. Šaltenis, and C. S. Jensen. Main-Memory Operation Buffering for Efficient R-Tree Update. In *VLDB*, 2007.
5. T. Brinkhoff. A Framework for Generating Networkbased Moving Objects. *GeoInformatica*, 6(2):153–180, 2002.
6. S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal Prefetching B+trees: Optimizing Both Cache and Disk Performance. In *SIGMOD*, 2002.
7. B. Cui, D. Lin, and K.-L. Tan. Towards Optimal Utilization of Main Memory for Moving Object Indexing. In *DASFAA*, 2005.
8. J.-P. Dittrich, P. M. Fischer, and D. Kossmann. AGILE: Adaptive Indexing for Context-Aware Information Flters. In *SIGMOD*, 2005.
9. J.-P. Dittrich and B. Seeger. GESS: a Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces. In *SIGKDD*, 2001.
10. Enhanced 911. `http://www.fcc.gov/pshs/911`.
11. Google Web Search. `http://www.google.com`.
12. G. Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35(1), 2006.
13. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.
14. D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.
15. P. Hough. Method and means for recognizing complex patterns. United States Patent No. 3069654, 1962.
16. H. V. Jagadish et al. Incremental Organization for Data Recording and Warehousing. In *VLDB*, 1997.
17. C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, 2004.
18. C. S. Jensen and S. Pakalnis. TRAX - Real-World Tracking of Moving Objects. In *VLDB*, 2007.
19. D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main Memory Evaluation of Monitoring Queries Over Moving Objects. *Distributed and Parallel Databases*, 15(2):117–135, 2004.
20. D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
21. G. Kollios, D. Papadopoulos, D. Gunopulos, and J. Tsotras. Indexing mobile objects using dual transformations. *VLDB Journal*, 14(2):238–256, 2005.
22. Kraftfahrt-Bundesamt. Number of Vehicles in Germany over time. `www.kba.de/Abt3_neu/FZ/Bestand/Themen_jaehrlich_pdf/bki1_2008.pdf`.
23. M.-L. Lee, W. Hsu, C. S. Jensen, et al. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003.
24. Loopt. `http://www.loopt.com`.
25. Apache Lucene. `http://lucene.apache.org/java/docs`.
26. K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD*, 2005.

27. K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous Nearest Neighbor Monitoring in Road Networks. In *VLDB*, 2006.
28. P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. The LHAM Log-Structured History Data Access Method. *VLDB J.*, 8(3-4):199–221, 2000.
29. P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4), 1996.
30. B. C. Ooi, K. L. Tan, and C. Yu. Frequent Update and Efficient Retrieval: an Oxymoron on Moving Object Indexes? In *WISE Workshops'02*, 2002.
31. J. A. Orenstein. An Algorithm for Computing the Overlay of k-Dimensional Spaces. In *SSD'91*, volume 525, 1991.
32. J. A. Orenstein and T. H. Merrett. A Class of Data Structures for Associative Searching. In *PODS*, 1984.
33. J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *SIGMOD*, 2004.
34. M. Pelanis, S. Šaltenis, and C. S. Jensen. Indexing the Past, Present, and Anticipated Future Positions of Moving Objects. *ACM TODS*, 31(1):255–298, 2006.
35. D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In *VLDB*, 2000.
36. F. Ramsak, V. Markl, et al. Integrating the UB-Tree into a Database System Kernel. In *VLDB*, 2000.
37. J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. *SIGMOD*, 29(2), 2000.
38. D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM TODS*, 1(3):256–267, 1976.
39. Personal communication with Skyguide Flight Control.
40. M. Stonebraker, S. Madden, et al. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, 2007.
41. Y. Tao, C. Faloutsos, et al. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In *SIGMOD*, 2004.
42. Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, 2003.
43. Y. Tao and X. Xiao. Primal or dual: which promises faster spatiotemporal search? *VLDB J.*, 17(5), 2008.
44. Tele Atlas MultiNet Europe Q4/2006. Germany.
45. D. Thirde et al. Evaluation of Object Tracking for Aircraft Activity Surveillance. In *2nd Joint IEEE International Workshop on VS-PETS*, 2005.
46. A. R. Thomas Legler, Wolfgang Lehner. Data Mining with the SAP Netweaver BI Accelerator. In *VLDB*, pages 1059–1068, 2006.
47. H. Tropf and H. Herzog. Multimensional Range Search in Dynamically Balanced Trees. *Ang. Informatik*, 23(2):71–77, 1981.
48. S. Šaltenis, C. S. Jensen, et al. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.
49. W. M. White, A. J. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling Games to Epic Proportion. In *SIGMOD*, 2007.
50. M. L. Yiu, Y. Tao, and N. Mamoulis. The Bdual-Tree: indexing moving objects by space filling curves in the dual space. *VLDB J.*, 17(3), 2008.
51. X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-Nearest Neighbor Queries over Moving Objects. In *ICDE*, 2005.
52. J. Zhou and K. A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *VLDB*, 2003.