

ETH ZURICH

ALGORITHMS FOR DATABASE SYSTEMS SEMINAR 2010

Indexing Moving Objects using Short-Lived Throwaway Indexes

A paper analysis by Willy Lai

laiw@student.ethz.ch

5/17/2010

Table of Contents

1. Introduction	2
2. Idea and cost comparison	3
3. MOVIES.....	4
3.1 Algorithm.....	4
3.2 Data structures.....	5
3.2.1 Update buffers.....	5
3.2.2 Indexes.....	5
4. Query Processing	6
4.1 Assumptions and Notations.....	6
4.2 Query enlargements	6
4.2.1 Predictive MOVIES.....	7
4.2.2 Non-Predictive MOVIES.....	8
5. Experiments.....	8
5.1 Goals and setup	8
5.2 Scalability in Index Size.....	9
5.3 Scalability in Update Rate.....	10
5.4 Shared-Nothing Scale-Out	10
6. Conclusion	11
References	12
6.1 Documents.....	12
6.2 Figures.....	12
7. Appendix.....	13
7.1 Multidimensional Indexes.....	13
7.1.1 k-d Trees.....	13
7.1.2 Z-Curve	14

1. Introduction

Traditional databases are dealing with the problem of persisting information over years and making it accessible for all kinds of users concurrently, while ensuring the ACID properties. The assumption, a lot of design decisions were based on, is that updates are relatively rare while queries are frequent. Let us consider a database for a discussion forum as an example. A user signs up once with his user name and password and uses these credentials on every login. The sign up boils down to a single insertion of a new row in the user table of the database. Upon every login a query is sent to the database for retrieving and comparing the credentials. Similarly a forum post is written once and read multiple times. This simple but fundamental assumption has led to sophisticated indexing structures being developed that are optimized for read operations.

The problem moving objects database are solving has additional requirements. For a given set of moving objects the database should efficiently retrieve the location of a moving object, such that the location returned does not differ by more than a given threshold from the actual location (having 100% accurate data for all moving objects at any time is clearly infeasible, as moving objects locations are changing continuously).

Besides of being query efficient, moving objects databases therefore need to support a reasonable high update rate. In addition to the current location, common techniques keep track of the speed and direction the objects are moving in. This has been shown to reduce the number of required updates in order to satisfy the threshold up to a factor of 3 [3]. However the support for high update rates remains the main challenge when dealing with moving objects. Most of today's indexing techniques used in moving objects databases that provide a mean to query for the location of cars, planes or any other moving targets using time parameterized queries, as the B^x-Tree [2], have evolved out of the traditional indexing techniques (see Appendix). As creation of an index has been considered costly, these techniques share the procedure that they are initialized once and maintained for the rest of the applications lifetime.

The analyzed paper criticizes this approach and proposes a new indexing technique for moving object databases that does not rely on a long lived index structure [1].

This report is structured as follows. Section 2 presents the idea of the proposed approach and how the authors have formally compared its costs involved against the costs of traditional “create once and update in place” strategy. Section 3 then elaborates the algorithm. Section 4 shows how query processing is done. Section 5 reveals the results of the experiments conducted in the paper. Section 6 will conclude the report eventually. There is no explicit analysis section as my commentaries and thoughts are embedded in every section. Additional material about two dimensional indexing can be found in the Appendix.

2. Idea and cost comparison

The presented paper proposes a novel main-memory approach for indexing, MOVIES (MOVing objects Indexing using frequent Snapshots), that uses short-lived throwaway indexes.

One key observation that precedes the core idea of MOVIES is that due to the high update rate of every single moving object, we receive a set containing new location data of every moving object in a relative short amount of time. If we wait for this set to be collected before applying any update we could build a complete new index out of the information contained in this set by bulk loading the data. This is the general idea of MOVIES. The authors' comparison against a general update-in-place strategy follows.

Given that we have an index structure that can handle multidimensional data with query cost $O(\log N)$ and bulk loading costs $O(N \log N)$. We use this structure to organize the spatial positions of our moving objects and to map these to the corresponding OIDs. Since an update on the index structure consists of a delete and an insert, we need two logarithmic traversals and therefore have an update cost¹ of

$$C_{update} = 2 c_1 \log_2 N$$

in the worst case, where c_1 is a system specific constant. For building a new index, which is done by collecting the data in a structure with cost c_3 in $O(1)$ and then bulk loading we get

$$C_{collect\ and\ rebuild} = C_{collect} + C_{bulkloading} = c_3 + c_2 N \log_2 N$$

where c_2 and c_3 are system specific constants. We now are interested in the number of updates W needed for the “collect and rebuild” strategy to be cheaper than the “update-in-place” strategy.

$$W c_3 + c_2 N \log_2 N \leq W 2 c_1 \log_2 N$$

$$W \geq \frac{c_2 N \log_2 N}{(2 c_1 \log_2 N - c_3)}$$

With estimated upper bounds for the systems constants $c_1 = 73.6ns$, $c_2 = 8.9ns$, $c_3 = 112.5ns$ and an index size of $N = 1'000'000$ the “collect and rebuild” approach is cheaper when $W \geq 62'872$.

We certainly have to mention that the comparison itself is an estimate. Besides estimating the constants it does not take into account the case when we do not receive updates from every object. It however does give a feeling on how MOVIES differs from update in place strategies and most importantly shows that we can trade staleness for better update performance while not touching the query costs.

¹ Note that the authors have assumed an index tree fan-out of 2 leading to the base 2 of the logarithm.

3. MOVIES

Now that we have grasped the idea of MOVIES this section is going to elaborate all steps of the algorithm and choices for its data structures.

3.1 Algorithm

The acronym MOVIES is not just an abbreviation, it also indicates on how one can conceive the algorithm, namely as a film reel, where each frame corresponds to a time interval in which following tasks are performed:

- Collect arriving updates in a buffer
- Build an index from the previous buffer
- Serve queries from the index built in the previous frame

The authors describe their algorithm using index frames F_i , which are active during a frame time $T_i = [t_i; t_{i+1})$. In each index frame a read-only index I_{i-1} that was build in the previous frame F_{i-1} answers all incoming queries, while an update buffer U_i collects all incoming updates during T_i . Parallel to that a new index I_i is built from the information collected in the update buffer U_{i-1} of the previous frame F_{i-1} and if necessary also from the index I_{i-1} (in the case the update buffer does not contain update from all OIDs, the missing locations need to be fetched from the previous index). As soon the new index has been built the next frame is started with an empty update buffer and the new index and the procedure is repeated. We will see that the old data structures can be reused and no new memory needs to be allocated.

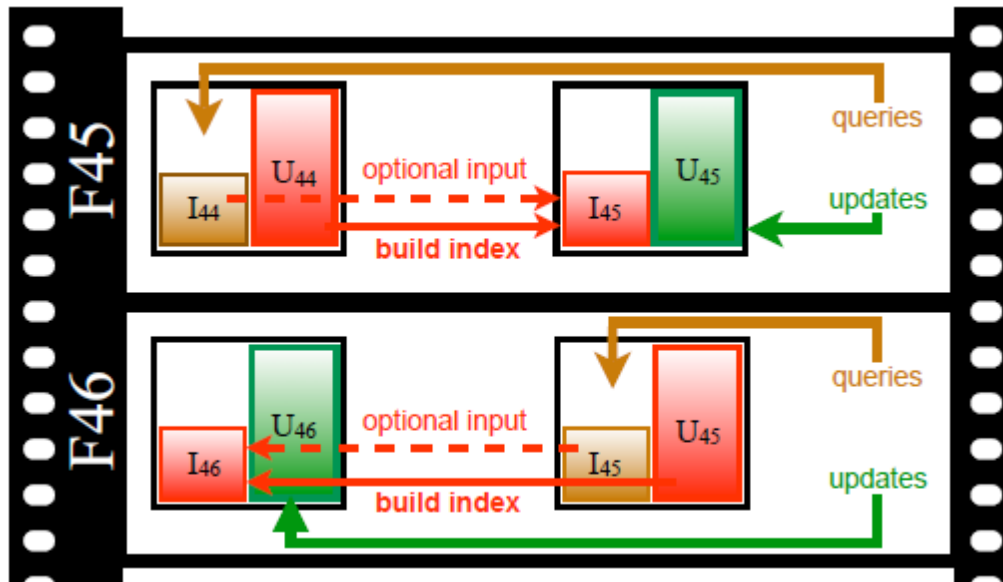


Figure 1: Illustration of an iteration of the algorithm of MOVIES

Let us illustrate that by going through a concrete example depicted from **Figure 1**. This figure shows the situation at frame F_{45} and F_{46} . At F_{45} queries are currently served from the index I_{44} (brown) that has been built in the previous frame while incoming updates are collected in the update buffer U_{45} (green) that has been initialized at the beginning of the current frame. While the index is traversed and updates are received a new index I_{45} (red) is bulk loaded from the location data stored in U_{44} (red) and I_{44} if necessary. Upon finishing the next frame F_{46} is started and the update buffer U_{44} as well as the index I_{44} can be cleared and reused as U_{46} and I_{46} respectively. Updates are now collected in U_{46} and queries answered from I_{45} .

As we can see the current index is built using the index which was one frame before the current frame and the update buffer which was two frames before the current frame. This is an important observation as we will see in section 4. The alert reader might have noticed that the frame time was defined as the bulk loading time in order to minimize staleness. However if the application can handle more staleness we could delay the next frame even longer and increase update performance.

3.2 Data structures

The following section elaborates the different data structures used for the MOVIES core algorithm.

3.2.1 Update buffers

The authors have implemented two versions of update buffers, FIFO update buffers (Logged MOVIES) and aggregated (Aggregated MOVIES) update buffers. FIFO update buffers are probably implemented using a linked list which allows the size to grow dynamically without wasting main memory space and still guaranteeing a constant insertion time. For high update rates however, multiple updates for the same OID can arrive during the same frame time. Storing several updates for the same OID in the update buffer is a waste of space because only the last update is relevant for constructing the new index. Aggregated update buffers counter this problem by keeping only the newest update of each OID in the queue. The authors have implemented aggregation buffers using arrays of size N , where N is the number of OIDs available. When a location update from OID i arrives the entry i in the array is overridden with the new location, ensuring that only one entry per OID is present. But this approach fixes the number of possible OIDs and only saves space when almost all moving objects send updates during the frame time. Otherwise array slots are left empty which are then unnecessarily allocated.

3.2.2 Indexes

The authors have chosen “linearized kd-tries” [6] to index their data. According to Donald E. Knuth a trie partitions the data space whereas a tree partitions the data. The name “linearized kd-tries” however seems to be contrived by the authors, since it does not appear in any referenced material. In any case it refers to data structures build on linearized multidimensional keys. The paper mentions that there is no need to construct the kd-trie at any time. This makes sense, as their throwaway indexes are read-only (indexes are built on a

single bulkload and not updated), which is another advantage of MOVIES, as we can opt for read optimized indexes and completely ignore the updateability. Therefore a sorted list of the composite keys suffices, since we can traverse the index in $O(\log n)$ using binary search.

4. Query Processing

This section describes how query processing is done in MOVIES.

4.1 Assumptions and Notations

For the remaining part of the report we use following assumptions and notations. The moving objects environment can be abstracted as a two dimensional square with $2^{16} \times 2^{16}$ possible coordinates. Moving objects are identified by a unique OID. Updates are sent as $(x, y, \overrightarrow{sv}, OID)$ -tuples where x, y are coordinates and $\overrightarrow{sv}$ is its speed vector, describing the speed and direction of movement. The moving objects are assumed to have a maximum speed S_{max} and to send updates at least every $t_{\Delta max}$ second.

Queries are time parameterized range queries $Q(r, t_q)$ with range (query window) $r = [x_{low}; x_{high}] \times [y_{low}; y_{high}]$ and query time t_q . $Q([8.505; 8.575] \times [47.350; 47.392], "09:00 CEST")$ for example queries for all OIDs that are near Zurich city at 9am (the exact format of the query parameters may vary).

4.2 Query enlargements

As we store our spatial data with respect to a single point in time, predictions need to be calculated for the location of those objects, whose timestamp differs from the timestamp t_q . One naive solution would be to

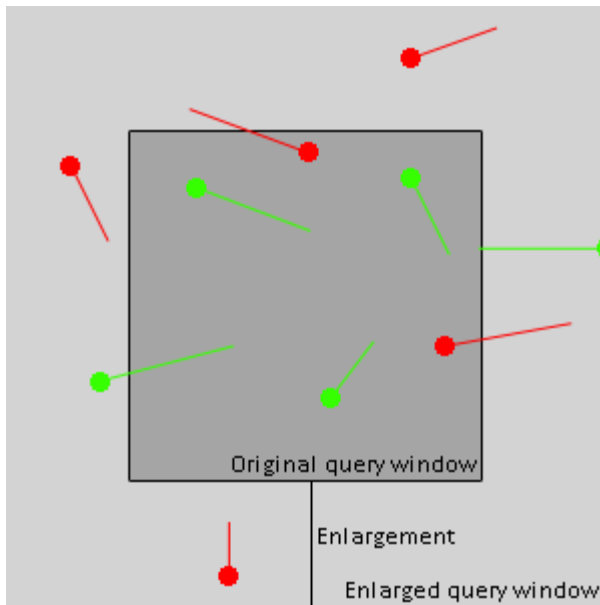


Figure 2: Illustration of a possible scenario for query enlargement.

extrapolate the locations for all moving objects to the query time t_q and query the range r on the extrapolated data. However the cost of a query would then become $O(n)$ where n is the size of the index and we may get false positives as well as false negatives.

The more efficient solution is to query the situation as it is and to enlarge the query window, so that it covers all moving objects that might possibly be in the original range at time t_q , this eliminates the false negatives but in general introduces more false positives. (False positives, in our case, are objects that are located in the query window at an earlier point in time, but have moved out of range at time t_q . Enlarging the

queried area will cover even more such cases). Another drawback is that a larger query window does negatively affect index traversing time, simply because a larger part of the index has to be traversed.

Figure 2 illustrates a possible scenario for query enlargements. The dark grey area indicates the original query window and the light grey area is the enlarged query window. The dots represent the current position of the moving objects and the line represents their movement and speed. At query time t_q all objects will be located at the end of their line. The objects of interest are marked green. These are the objects that will be located in the original query window at time t_q . All other objects are marked red.

First of all we see that at the current time the original query window does not cover all objects that the query asks for, also does it contain objects that will not be located in the area at time t_q . In order not to have false negatives, we need to enlarge the query window such that it covers all green objects. This of course may introduce additional false positives.

The authors propose two possible ways (Predictive MOVIES and Non-Predictive MOVIES) of how these enlargements can be done. Both strategies are independent of the data structure used for the update buffer, resulting in four different versions of their MOVIES implementation.

4.2.1 Predictive MOVIES

In Predictive MOVIES all location information in the index correspond to the same point in time $t_{index} > t_{u_i}$ where t_{u_i} is the time when the update u_i arrived. Every position update is transformed to the predicted position $(x, y) + \vec{sv}(t_{index} - t_u)$ at time t_{index} (Moving objects that did not receive an update during the frame time, are transformed the same way using $t_u = t_{old\ index}$). This follows from the assumption, that the moving objects are not changing their direction and speed between two consecutive updates, which is a reasonable assumption for a small $t_{\Delta max}$. Predictive MOVIES does not need additional space to store t_u along with the position, as only the predicted position has to be kept in the update buffer. The drawback of this method is the additional processing cost per incoming update.

A query $Q(r, t_q)$ is answered by querying the index for the range $[x_{low} - \varepsilon; x_{high} + \varepsilon] \times [y_{low} - \varepsilon; y_{high} + \varepsilon]$ where $\varepsilon = S_{max} \cdot |t_q - t_{index}|$. The assumptions behind these transformations is that a moving object might have moved during $t_q - t_{index}$. In the worst case it was located at the border of r and was moving in the perpendicular direction away from the rectangular query area with the maximum speed S_{max} .

In order to minimize the needed enlargement the authors choose the t_{index} that minimizes the sum of the added area over all queries $q_1 \dots q_k$.

$$\sum_{i=1}^k \left[\left((x_{high}^i + \varepsilon) - (x_{low}^i - \varepsilon) \right) \cdot \left((y_{high}^i + \varepsilon) - (y_{low}^i - \varepsilon) \right) - (x_{high}^i - x_{low}^i) \cdot (y_{high}^i - y_{low}^i) \right]$$

or alternatively $\sum_{i=1}^k |t_q^i - t_{index}|^2$. They assume that queries never ask for past times i.e. $t_q = t_{now} + t_\Delta$ (where t_{now} is the relative current time of the incoming query)², which is the case for most real world applications. Recall from section 3.1 that the update buffer initialized at t_i is used for building the index active at $t_i + 2T_{frame\ time}$, which sets the lower bound of t_{index} . Assuming a uniform distribution of the query arrival time t_{now} within a frame, the t_{index} is preferably set as the mean of t_{now} i.e. shifted $+0.5T_{frame\ time}$. This is optimal for query times $t_q = t_{now}$ but we need to consider Δt , since queries might ask for times in the future, so we end up with $t_{index}^i = t_i + 2.5T_{frame\ time} + \Delta t$.³

4.2.2 Non-Predictive MOVIES

In Non-Predictive MOVIES each position update is stored with the timestamp of its arrival. This avoids having to perform additional computations on every update, but instead needs more storage space, as the timestamp of each update must be kept in the buffer.

Query enlargements are done as in Predictive MOVIES except that $\varepsilon = S_{max} \cdot |t_q - t_{min}|$ where t_{min} is the arrival time of the oldest entry in the index. According to the authors, finding t_{min} comes with prohibitive costs. For this reason they use $t_{min} = t_{now} - (t_{\Delta max} + 2T_{frame\ time})$ (t_{now} is the time where the update buffer was initialized), which is a lower bound for the time of the oldest update. Again these enlargements come with the disadvantage of introducing additional false positives.

The reasoning about the lower bound is simple. Once more we make use of the observation in section 3.1 and derive that it takes at most $2T_{frame\ time}$ for an update to appear in the index. Also, do we have to account for the fact, that moving objects send updates only every $t_{\Delta max}$. If $t_{\Delta max} > T_{frame\ time}$ it could happen that for a given OID no entry will be in the update buffer and the timestamp from the old index has to be used. Note that if $t_{\Delta max} < T_{frame\ time}$ the lower bound could be simplified to $t_{min} = t_{now} - t_{\Delta max}$.

5. Experiments

In the section 2 we have seen, that the MOVIES approach can indeed outperform update in place strategies. This section goes through the experiments conducted in order to evaluate MOVIES in practice.

5.1 Goals and setup

The authors listed their goals of their experiments as

² Be aware that the authors use t_{now} to refer to current time for issued queries, but also for the starting time of the current frame. At least that is what I observed. Otherwise the formulas for t_{index} in section 3.5.1 and t_{min} in section 3.5.2 would make no sense to me.

³ Note, that there is an additional index i on t_{index}^i which indicates the frame identifier, as the absolute t_{index} is different for each frame. The authors use $t_{index} = t_{now} + 2.5T_{frame\ time} + \Delta t$, which is ambiguous since t_{now} is also used to refer to the current time.

1. Determining the maximum supported update rate of MOVIES when scaling the index size (section 5.2)
2. Determining the query throughput of MOVIES when scaling the update rate (section 5.3)
3. Determining the performance of MOVIES when implemented on a cluster of shared nothing machines (section 5.4)

For their experiment they used a real road network of Germany covering 38 million nodes and 40 million road segments and assumed $S_{max} = 216 \frac{km}{h}$. Moving objects were generated and placed uniformly distributed on the nodes of the network and assigned a constant random direction which they are trying to follow on the network. Over all experiments the data and indexes were completely in main memory. Queries are time-parameterized range queries with $t_q = t_{now} + 1.5t_{\Delta max}$ and a range of roughly $1km \times 1km$, where the center was chosen the same way as the placement of the moving objects.

The experiments were run on identical machines with 4 cores 6 GB of main memory each. Two machines were used to generate updates and queries, and up to 4 machines were used to process them. In order to tap the full potential of the machines, the authors implemented a multi-threaded variant of MOVIES. Besides of MOVIES the authors also conducted the same experiments with other indexing structures like binary search trees, B⁺-trees and B^x-trees, all these index structures were parallelized by partitioning the OID space to each core and indexing them separately. Unfortunately they do not disclose any information on their implementation (e.g. what linearization they use for the tree-based structures and how they implemented the multi-threaded variant of MOVIES), which leaves us with no choice but to trust them on keeping the comparison fair.

5.2 Scalability in Index Size

In this experiment the authors kept a fixed query rate of $1000 \frac{queries}{s}$ and measured the maximum supported update rate by varying the size of the index. The binary search tree did only manage to handle an index size of 3.2M objects while the B⁺-tree and the B^x-tree could scale up to 12.8M moving objects in the index. All MOVIES variants performed much better as they all could handle index sizes up to 25.6M and achieved constantly better update rates than all tree-based methods.

Comparing the four different Movies Variants showed, that the prediction calculation at indexing time is fairly costly as the throughput measured from the PI versions were only half as much as the results from the NPI versions. Logged versions seem to perform slightly better than aggregated versions, which is probably due the larger cost factor for insertion in aggregated buffers. Aggregated buffers however perform better in respect to staleness. Part of that can be accounted from the shorter build time of the index when using aggregated buffers. A shorter index build time implies a shorter frame time and therefore a higher index rebuild

frequency. However I'm surprised of the result, that the average staleness can grow up to **19 sec** for logged NPI and only **7 sec** for aggregated NPI.

5.3 Scalability in Update Rate

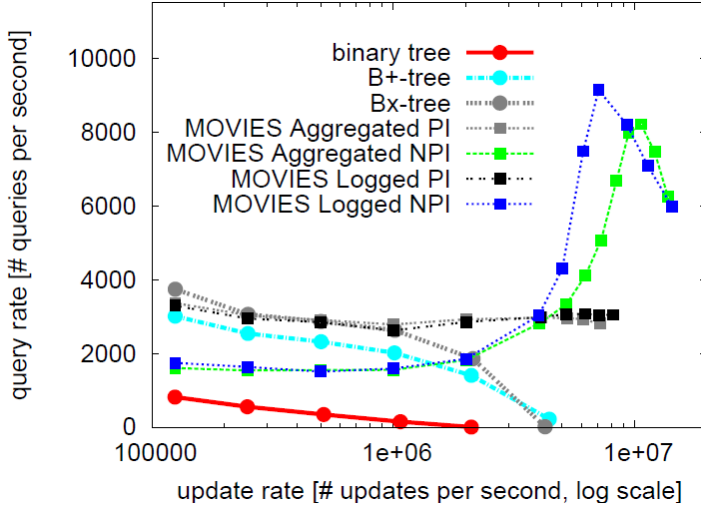


Figure 3: Results of the measurements done in the query-rate vs. update rate experiment.

For this experiment the index size was kept constant at **6.4M** objects and measured the query rate when scaling the update rate. Again all tree based methods perform relatively bad against MOVIES. One interesting observation can be depicted from Figure 3 which is taken from the paper. Up to $4M \frac{\text{updates}}{s}$, PI versions of MOVIES achieve higher query rate than NPI versions. But for higher update rates, the query performance sharply increases up to $9200 \frac{\text{queries}}{s}$. The explanation is that for lower

update rates $t_{\Delta max}$ is larger, which has the consequence that query enlargements is big-

ger for NPI versions, since they take into account the time of the oldest possible update. PI versions perform better as they do not depend on a t_{min} . For higher update rates however $t_{\Delta max}$ will decrease to a point, where the size of window enlargement for NPI versions will be approach the size of the query enlargement done in PI versions without having the additional overhead in collecting the updates. Another noteworthy outcome is that as expected, staleness was relatively high for low update rates, because not all OIDs might have sent location updates when building a new index. Consequently old date from the previous index has to be used in the new index.

5.4 Shared-Nothing Scale-Out

In order to test scale out on shared-nothing machines, the OIDs were horizontally hash-partitioned. Index size was kept constant while the processing nodes where varied from 1 to 4, while for each the maximum update rate is measured. The results show a linear increase in update rate and, when keeping the update rate constant, a linear decrease in staleness for all MOVIES variants, meaning that all MOVIES variants scale very well.

6. Conclusion

The idea of rebuilding indexes rather than maintaining them might seem unprofitable at first. This paper proves the exact opposite and shows that constant factors in cost calculations do matter. The MOVIES approach trades data staleness for query performance and we observe that the resulted staleness is minor and can be easily handled by most application scenarios while the improvements in update performance are immense given a reasonable sized number of moving objects. I was surprised in how much better the MOVIES approach performed compared to other indexing structures, which relied on maintaining an index. But it has to be said, the comparisons are not completely transparent, leaving some open questions.

Besides that, only some details in their formulas and presentation of measurement results (e.g. no standard deviations or confidence intervals given) can be faulted.

Overall the paper leaves a solid impression on me. The authors explained their algorithm and their choice of data structure in a clean way and used reasonable assumptions for their design decisions as well as for their experiments. Choosing a workload that reflects real world application was crucial, as for smaller workloads MOVIES might not have excelled.

References

This section contains text references and figure references. Figures have been created by myself if not mentioned otherwise.

6.1 Documents

- [1] J. Dittrich, L. Blunski, M. A. Vaz Salles. Indexing Moving Objects Using Short-Lived Throw-away Indexes. SSTD 2009
- [2] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In VLDB, 2004.
- [3] A. Civilis, C. S. Jensen, J. Nenortaitė, and S. Pakalnis. Efficient Tracking of Moving Objects with Precision Guarantees. In Proc. MobiQuitous, 2004
- [4] A. Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching, Proc. 1984 ACM SIGMOD International Conference on Management of Data, pp. 47–57
- [5] J. L. Bentley. Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9):509-517, 1975.
- [6] H. Tropf and H. Herzog. Multidimensional Range Search in Dynamically Balanced Trees. Ang. Informatik, 23(2):71–77, 1981.

6.2 Figures

- Figure 1: Illustration of an iteration of the algorithm of MOVIES 4
Source: Analyzed paper "Indexing Moving objects using Short-Lived Throwaway Indexes"
- Figure 2: Illustration of a possible scenario for query enlargement..... 6
- Figure 3: Results of the measurements done in the query-rate vs. update rate experiment..... 10
Source: Analyzed paper "Indexing Moving objects using Short-Lived Throwaway Indexes"
- Figure 4: Illustration of k-d-Tree 13
Source: Created by Jens Teubner for lecture slides on "Architecture and Implementation of Databases"
- Figure 5: Illustration of the ordering resulting from key concatenation 14
- Figure 6: Illustration of the locality preserving Z-order 14

7. Appendix

This section covers material that I assume is prior knowledge for most readers. For completeness however I give an overview that is sufficient for the scope of the report. For more detailed information consult the references.

7.1 Multidimensional Indexes

Indexes on one dimensional data lose their value, when querying data of other dimensions. In order to take into account multiple dimensions we could build a separate index for each dimension. But we would need to traverse every index and perform an intersection of the results which is relatively expensive. Also does keeping multiple indexes increase update and maintenance cost by factor n , where n is the number of indexes.

Instead we want to keep a single index that is designed for multidimensional data, such as a k-d Trees or an R-tree [4]. Another possible solution is to linearize the keys of the multidimensional data into a one dimensional code, and index these codes in a one dimensional index structure, as is done in B^x-trees [2].

7.1.1 k-d Trees

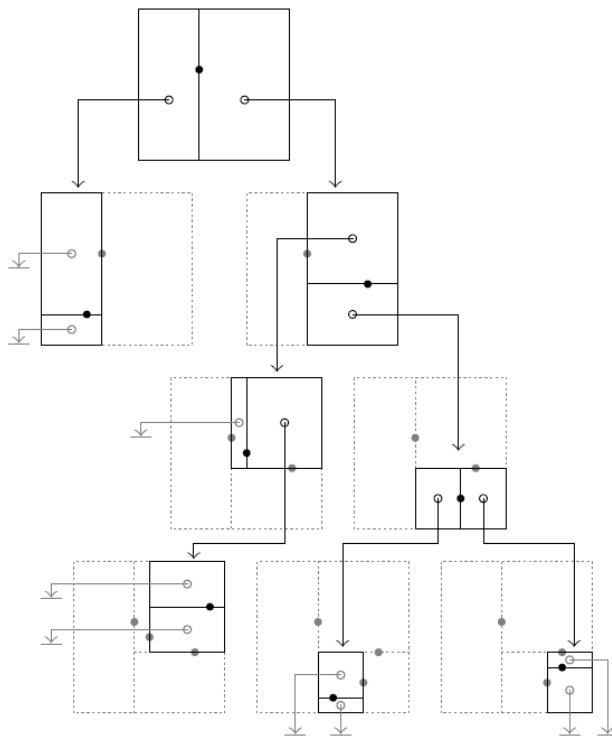


Figure 4: Illustration of k-d-Tree

A k-d tree [5] is a binary tree which partitions a k-dimensional space. Every node corresponds to a k-dimensional point in space. A binary structure is achieved by partitioning each node in the tree by only one dimension (unlike 2^k -ary trees where each node in the tree is partitioned into 2^k regions). Each tree level is labeled with a discriminator which indicates the dimension along which the node is partitioned. A k-d tree can be constructed by iteratively inserting the median of all values to be inserted into the current node (sub tree) with respect to the dimension in which the node is to be split up. An illustration of a two-dimensional k-d tree can be depicted from Figure 4.

One disadvantage making k-d trees not suited for moving object databases is the fact, that there are no techniques known to re-balance k-d trees, leading k-d trees to degenerate sooner or later in an environment where a lot of updates take place.

7.1.2 Z-Curve

Linearization in our sense is the process mapping multiple values into a single value, which allows us to define a clear ordering of the multidimensional data. A simple linearization would be to concatenate the coordinate values and use the value of their concatenation for indexing. However the composite generated would be primarily sorted according to one dimension, making the index inefficient for queries on other dimensions.

The Z-Curve [5] is a space-filling curve, generated by interleaving the binary representation of the dimension values. This generates a composite key that is much more symmetric, meaning that it treats the dimensions in a fairer way in respect to order priority, than when using concatenation of the values. The two-dimensional case for both linearization strategies is illustrated in the following figures

	000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
000 0	000000 0	000001 1	000010 2	000011 3	000100 4	000101 5	000110 6	000111 7
001 1	001000 8	001001 9	001010 10	001011 11	001100 12	001101 13	001110 14	001111 15
010 2	010000 16	010001 17	010010 18	010011 19	010100 20	010101 21	010110 22	010111 23
011 3	011000 24	011001 25	011010 26	011011 27	011100 28	011101 29	011110 30	011111 31
100 4	100000 32	100001 33	100010 34	100011 35	100100 36	100101 37	100110 38	100111 39
101 5	101000 40	101001 41	101010 42	101011 43	101100 44	101101 45	101110 46	101111 47
110 6	110000 48	110001 49	110010 50	110011 51	110100 52	110101 53	110110 54	110111 55
111 7	111000 56	111001 57	111010 58	111011 59	111100 60	111101 61	111110 62	111111 63

Figure 5: Illustration of the ordering resulting from key concatenation

	000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
000 0	000000 0	000001 1	000100 4	000101 5	010000 16	010001 17	010100 20	010101 21
001 1	000010 2	000011 3	000110 6	000111 7	010010 18	010011 19	010110 22	010111 23
010 2	001000 8	001001 9	001100 12	001101 13	010000 16	010001 17	010100 20	010101 21
011 3	001010 10	001011 11	001110 14	001111 15	010010 18	010011 19	010110 22	010111 23
100 4	100000 32	100001 33	100100 36	100101 37	110000 48	110001 49	110100 52	110101 53
101 5	100010 34	100011 35	100110 38	100111 39	110010 50	110011 51	110110 54	110111 55
110 6	101000 40	101001 41	101100 44	101101 45	110000 48	110001 49	110100 52	110101 53
111 7	101010 42	101011 43	101110 46	101111 47	110010 50	110011 51	110110 54	110111 55

Figure 6: Illustration of the locality preserving Z-order

Z-codes are locality-preserving, as they group the multidimensional data points, which are near to each other (in a geometrical sense) into clusters, making them much more efficient for range queries. Z-Codes are similar to k-d trees in the sense that they alternate the dimensions in their indexing scheme. The main difference however is that Z-Codes structure the data space, while k-d trees structure the data. This difference is very important, as we can really treat the codes like one-dimensional keys over the z-order and can use one-dimensional self-balancing data structures like B⁺ trees for indexing them.