

Algorithms for Database Systems

ST²B-tree: A Self-Tunable Spatio-Temporal B⁺-tree Index for Moving Objects

Report by: **Lucia Ambrošová**
Supervisor: **Prof. Dr. Peter Widmayer**
Shepherd: **Georgios Giannikis**

May 31, 2010

Contents

1	Introduction	1
2	Background Concepts	2
2.1	Database Indexing and the B ⁺ -Tree	2
2.2	Moving Object Databases	2
3	Previous Work	3
3.1	TPR*-Tree	3
3.2	B ^x -Tree	3
4	Challenges in Moving Object Indexing	4
4.1	Research Objectives	4
5	ST²B-Tree	5
5.1	Self-Tuning of the ST ² B-Tree	6
6	Performance Evaluation	8
6.1	Spatial Diversity	8
6.2	Temporal Diversity	9
6.3	Spatio-Temporal Diversity	9
6.4	Throughput Test	10
7	Conclusion	10
	References	11

1 Introduction

This report summarizes the challenges in creating indexes for Moving Object Databases (MODs) and presents the Self-Tuning Spatio-Temporal B^+ -Tree (ST^2B -Tree) developed in [1] as an answer to these challenges. Before doing so, however, it provides a concise introduction of necessary background concepts and reviews previous work in the field. Let us begin the adventure.

Moving Object Databases store information about moving objects - including their coordinates in space. As these have to be searched efficiently, there is a need to maintain an index. However, spatial coordinates are subject to frequent updates, thus the index has to be capable of performing updates very quickly and without blocking other updates.

Additionally to these constraints, an index on spatial coordinates is expected to preserve so called spatial locality. This means that objects which are close to each other in space can be found close to each other in the index. Spatial locality facilitates typical MOD queries such as range queries, where the user would like to retrieve all objects within a certain spatial range. The need for this feature has been recognized and accounted for by several previous solutions. [1] classifies the most successful approaches into two categories. Superficially, the first one partitions spatial objects hierarchically using Minimum Bounding Rectangles (MORs), where each index node bounds all the contained objects by the smallest possible rectangle. The most representative example from this category is the TPR^* -tree introduced in [4]. The second approach with a representative B^x -tree developed in [2] partitions space into a 1D representation using space filling curves to achieve approximate spatial locality. Even though the TPR^* -tree has a nicer model of spatial locality, it generally under-performs the B^x -tree quite radically due to high maintenance costs. Both concepts are explained in more detail and contrasted in section 3.

Being the more efficient option, the B^x -tree index model is much favored by the authors of [1]. However, as opposed to the TPR^* -tree, the B^x -tree does not account for the unevenness of data in space and with time. The model relies on even partitioning of space, which might often result in workload imbalance across the index and consequent performance degradation. The novel contribution of [1] is to identify 3 main types of data diversities that may compromise the performance of the B^x -tree. The authors then propose the ST^2B -Tree as a solution that accommodates for such diversities and minimizes performance degradation.

In real world moving object examples it is not uncommon for objects to be arranged more densely in some places (hotspots) and sparsely in others. At the same time the total number of objects in the database might change quite dramatically with passing time. Finally, the hotspots may also move, vanish and new ones can arise. The three above mentioned phenomena are referred to as diversity in time, space and space-time, respectively.

The ST^2B -tree partitions space based on the current object densities at each time step. It distinguishes hotspots from sparse regions and finds suitable 1D representation for each region individually with the goal of preventing overflow index nodes. The regions are periodically adjusted to reflect the current object distribution. Using these two main ideas the ST^2B -tree is equipped to handle all three data diversities.

Finally, the ST^2B -tree has been experimentally demonstrated to be superior to the B^x -tree and TPR^* -tree in situations with dramatically uneven and continuously changing object distributions. The detailed experiment results are analyzed in section 6.

2 Background Concepts

2.1 Database Indexing and the B⁺-Tree

Retrieval of records based on the value of a specific attribute is often facilitated by the use of indexes. To prevent having to scan the entire table, which can be done by binary search, but only in case the records are sorted by the attribute, we build an additional structure. The index structure allows quick access of elements based on the attribute value.

The most common index categories are hash and tree indexes. In a hash index, the hashed attribute value points to a 'bucket' with all corresponding records. In case of a range query, each value in the range has to be explored individually. On the other hand, tree indexes keep values in a sorted manner, which eases range queries.

The most typical tree index is the B⁺-tree, an example of which is shown in Figure 1 below. The leaf nodes contain pointers to the physical records. They are sorted by the attribute value and chained, thus they can be traversed to retrieve records corresponding to a range of values. Internal nodes are used to navigate to the correct leaves.

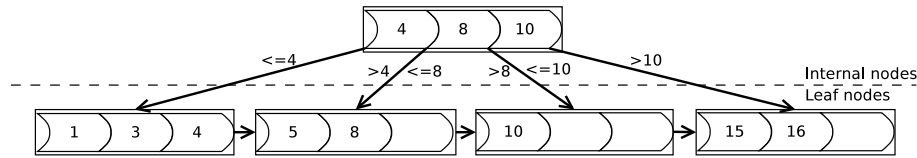


Figure 1: B⁺-tree structure

Importantly, insert and delete operations on a B⁺-tree always leave the tree balanced, which means that the route to each leaf is of the same length [3]. This property made the B⁺-tree one of the most popular indexes that is nowadays present in virtually all commercial databases.

2.2 Moving Object Databases

Moving Object Databases have applications in many areas: from taxi-cabs through digital battlefield in military to meteorology monitoring and many others. Typical MOD queries in all these systems are “Find all objects which will be in a certain range in m minutes” or similarly “Find the k objects which are closest to a certain point”. These types of queries are referred to as predictive range queries and kNN (k nearest neighbors) queries, respectively.

Storing coordinates of continuously moving objects is a task beyond the capabilities of regular databases [5], where data is assumed to be constant unless explicitly updated. In case of moving objects, assuming static locations until the next update would result in a crude misrepresentation of object positions between updates. Therefore, MODs combine each object's velocity and position at update time to estimate its position at any time until the next update. Nevertheless, predicted location is still inherently imprecise.

One of the biggest MOD challenges is how to index dynamic attributes such as object locations. More specifically, how to represent multidimensional space and map objects and queries into this space. Typically, a MOD receives many more location updates than queries [1]. Therefore an acceptable MOD index has to be primarily able to support fast concurrent updates. Fast query execution should come as a result of good space representation.

3 Previous Work

3.1 TPR*-Tree

The TPR*-tree partitions objects in multidimensional space by a hierarchy of Minimum Bounding Rectangles (MBRs). Figure 2a shows the mapping of objects a through j into a TPR*-tree.

Each object has a position and velocity. The MBRs consider the velocities of the objects they encapsulate to derive their own velocity in each direction. In the example from Figure 2a, if objects a and d move away from each other, the walls of the enclosing MBR N_5 will move apart.

In order to answer a range query, the evaluation makes its way down the tree hierarchy, following an internal node only if the query range overlaps with the MBR. Since the objects and MBRs are dynamic, predictive queries are answered using MBR enlargement, where the size of each MBR is estimated at query time.

The benefit of MBR enlargement is that each MBR is enlarged separately based directly on the speeds of all encompassed objects, which means that they are not inflated to unnecessary sizes based on some 'maximum speed' constant as in other models. Consequently, fewer MBRs are likely to overlap the query region, leading to faster execution.

The structure of the TPR*-tree makes it highly adaptable to changes in data density in space and time. However, updates are extremely expensive, yielding a high maintenance overhead.

3.2 B^x -Tree

The B^x -tree exploits the widespreadness and efficiency of the B^+ -tree (outlined in section 2.1) upon which it is based. 2-dimensional space is linearized using space-filling curves, which enumerate every point in a discrete multi-dimensional space. Spatial discreteness is achieved by simply partitioning space using a uniform grid.

Arranging the index purely by position without taking into consideration the temporal dimension would paradoxically destroy the spatial locality property.

Example: Consider two objects updating their coordinates to the same grid cell at two different times. The mapping would put them next to each other in the index, however by the time the second object updated its coordinates, the first one could be in a different part of the grid.

Therefore, the B^x -tree splits the index into several subtrees, where each one contains objects updated within its small time interval, arranged by location. The index wraps around with time.

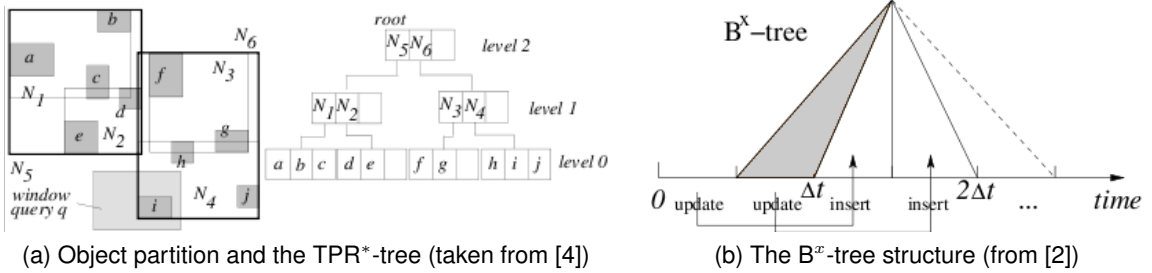


Figure 2: The TPR*-tree and B^x -tree structure

At every point the old part of the tree is being emptied and the current one is being filled with objects as shown in Figure 2b above.

As mentioned above, the advantage of the B^x -tree comes primarily from the usage of the underlying B^+ -tree which has much lower maintenance costs compared to the TPR*-tree. On the other hand, operating with a constant grid throughout the space and time makes the B^x -tree completely oblivious to the diversity of object distributions in both space and time.

4 Challenges in Moving Object Indexing

This section explores the impact of grid granularity on the performance of a B^x -tree and the 3 types of data diversities discussed in [1].

In a grid that is coarse relatively to the object density, each cell contains many objects, all of which have to be checked during a query evaluation. There is a high number of false positives that need to be eliminated. On the other hand, with a relatively fine grid, each query window maps to a large number of 1D range queries that have to be checked individually. Both underestimating and overestimating the grid size therefore contributes to performance deterioration.

Density of moving objects expectedly vary across different areas in space and with time. Consider a database storing information about taxi-cabs. The central city areas are likely to act as hotspots daily during rush hours. At the same time, residential areas might be considerably sparser with fewer people around to be interested in taxi services. Similarly, the total number of taxis on the road at night is most likely much lower than during day time. Finally, the location, number and density of individual hotspots might also vary with time such as the residential areas becoming more crowded again after regular office hours.

Using a uniform grid across space and time therefore results in overestimating the necessary grid size in some areas while underestimating it in others. Such suboptimal behavior may significantly degrade the index performance. The authors of [1] thus point out that it is vital to adjust grid sizes to varying object densities in space and continuously reevaluate the grids with time. The specific objectives of their research efforts are summarized in the following section.

4.1 Research Objectives

The performance of an index is compromised if it partitions space evenly using a uniform grid size. The following two criteria were formulated to distinguish a good moving object index:

1. Discrimination between regions of different densities,
2. Adaptation to density and distribution changes with time.

The resulting index then qualifies as a self-tuning spatio-temporal index. Note that [1] implicitly assumes a number of other constraints such as:

- Ease of implementation
- Support by many commercial databases
- Fast insert/delete operations in a concurrent environment

As all these criteria are satisfied by the B^+ -tree, it is an ideal basis for a good index.

5 ST²B-Tree

The Self-Tunable Spatio-Temporal B⁺-tree index (ST²B-tree) is built directly on the B⁺-tree without requiring any structural or algorithmic changes. Each moving object's key in the B⁺-tree is derived from a combination of the object's latest update time and position.

Similarly to the B^x-tree (see section 3.2), the ST²B-tree is logically split into 2 subtrees BT_0 and BT_1 corresponding to two consecutive time intervals of length T . Each object is indexed in the subtree covering the object's latest update time. At every point in time one of the subtrees covers an expired time period and is being emptied while the other subtree fills up with current updates. The index thus rolls over as indicated in Figure 3. Subtree affiliation determines the temporal component of an object's key $KEY_{time} \in \{0, 1\}$.

The multidimensional space contains n reference points $RP_1, \dots, RP_n$ which define disjoint regions $VC_1, \dots, VC_n$ encompassing all points closest to each individual reference point¹. Each cell VC_i has its own grid G_i with a granularity that reflects the density of objects within the cell. A portion $SPAN_{space}$ of keys is allocated to each grid. The cells within every grid are enumerated using the Hilbert space filling curve that approximates spatial locality. The spatial key of an object o located at point $\vec{x}$ with velocity $\vec{v}$ within a grid G_i is then determined as $KEY_{space}(o) = i \times SPAN_{space} + cell_id(\vec{x}', G_i)$. The complete key is thus

$$KEY_{ST^2} = KEY_{time} \times SPAN_{time} + i \times SPAN_{space} + cell_id(\vec{x}', G_i).$$

Note that in order to approximate spatial locality within the ST²B-tree, all objects positions are indexed as the estimated positions $\vec{x}'_i$ at the end of the corresponding subtree's interval T_{ref} . Thus, $\vec{x}' = \vec{x} + \vec{v} \cdot (T_{ref} - t_{up})$ where t_{up} is the last update time of the object.

The index rolls over periodically after time T . Assuming that every object is updated at least once within each period, the old subtree is empty at the rollover time. At this point the old portion of the tree becomes the current one and vice versa. The ST²B-tree uses the switching moment to produce a new set of reference points and corresponding grids to accommodate for changes in object distribution. Consequently, the two subtrees have different reference points.

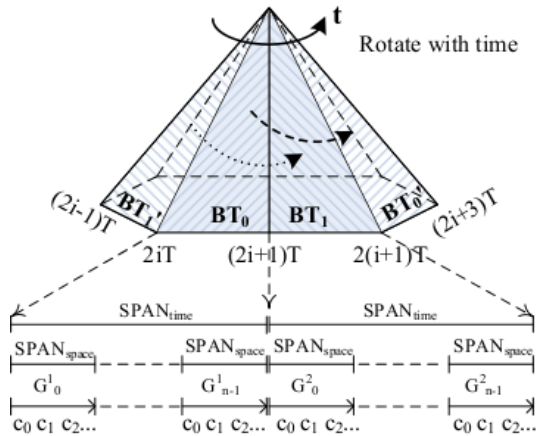


Figure 3: The ST²B-tree structure ([1])

Algorithm 1 Range Query

```

result = 0
for each subtree  $BT_i$  do
   $R'_q = EnlargeRegion(R_q, max_v, t_q, T_{refi})$ ;
  for each reference point  $RP_j$  do
    if  $R'_q$  intersects with  $VC_j$  then
      for each cell of  $G_j$  intersecting  $R'_q$  do
        for each object  $o$  in the cell do
           $p_{t_q} = Position(o, t_q, T_{refi})$ ;
          if  $p_{t_q} \in R_q$  then
            result = result  $\cup \{o\}$ 
return result

```

Figure 4: The range query algorithm

Evaluating a query over the spatial range R_q at some future time t_q requires searching each subtree separately. In contrast to the TPR*-tree (see section 3.1) which enlarges individual

¹These are the Voronoi cells of the reference points

MBRs based on the contained objects' speeds to represent the state of the system at query time, the ST²B-tree algorithm relies on query window enlargement, i.e. extending the query range to cover all objects indexed at T_{refi} that can be in R_q at time t_q .

Example: Let us consider a subtree that expires at time $T_{refi} = 5$ in a system with a maximum speed $max_v = 4$. Imagine we would like to retrieve all objects within a region $R_q = ((x_1, y_1), (x_2, y_2))$ at time $t_q = 3$. All objects are indexed at time $T_{refi} = 5$ and since time $t_q = 3$ they could have traveled at most $max_v * (T_{refi} - t_q) = 4 * 2 = 8$ distance units. An object within R_q at time 3 can then be located at most 8 units away from R_q at time 5. Therefore, the query window has to be extended to $R'_q = ((x_1 - 8, y_1 - 8), (x_2 + 8, y_2 + 8))$.

The downside of this approach is that the query range is enlarged based on the maximum speed in the system max_v and is thus likely to produce excessive numbers of false positives that have to be considered and eliminated individually. On the bright side, compared to MBR enlargement, query window enlargement is much faster because it is only performed once.

Once the query window is enlarged to R'_q , the algorithm finds all regions VC_j that intersect the enlarged range R'_q and proceeds to do the same with individual grid cells of G_j . All objects within matching grid cells are subsequently examined and are added to the result set only if their estimated positions at time t_q are within R_q . The pseudocode for the range query algorithm is provided in Figure 4. A kNN query (see section 2.2) on the ST²B-tree is evaluated by incrementally running range queries until exact k nearest neighbors are found.

5.1 Self-Tuning of the ST²B-Tree

The ST²B-tree structure accommodates for data distribution in two ways. Firstly, grid granularities are tuned to fit local object densities and secondly the reference points and grids are periodically updated. This section details the tuning strategies proposed in [1].

Since a typical MOD has to process a large number of updates per second, it is unthinkable for the tuning process to stall all other DB operations for extended time periods. It is thus vital to use efficient algorithms. At the same time, optimality of the resulting structures is not strictly required as it would anyway be very short-lived in the highly dynamic MOD setting.

The tuning procedure relies on a 2D histogram capturing object distribution in space. The histogram consists of $n \times n$ square-sized cells, each storing the estimated number n_{ij} of objects within it at time $T_{ref} + T$ (the reference time of the upcoming subtree) along with the centroid $\vec{x}_{ij}$ of these objects. New reference points are established by identifying sparse and dense regions in the histogram. The algorithm used is region growing (see pseudocode in Figure 5) which considers two neighboring cells as belonging to the same region only if they are similar, i.e. if they are expected to enclose a similar number of objects at time $T_{ref} + T$.

Even though the authors of [1] do not mention the computational overhead associated with maintaining the histogram, it expectedly needs to be updated at every object position update. This would imply an additional write operation at each update.

The region growing algorithm takes the previous subtree's reference points as seeds. Object densities in space change gradually, thus the new regions are likely to only change slightly. The algorithm starts with the grid cells that house the old reference points and recursively examines all adjacent cells. A neighbor cell c' is added to a region if it hasn't been assigned to a different region and the number of objects it includes $c'.n$ is within a small error margin ϵ from the region's cells' minimal and maximal number of objects $R.min_n, R.max_n$.

Algorithm 2 Region Growing	
$RS = \emptyset$ for each previous reference point RP_k do $c =$ the cell that contains RP_k if c is unmarked then Add c to a new region R ; mark c ; $Growing(c, R)$; Add R to RS while there is c that is unmarked do Add c to a new region R ; mark c ; $Growing(c, R)$; Add R to RS return RS ;	Function $Growing(c, R)$ for each neighbor cell c' of c do if c' is unmarked and $ c'.n - R.max_n \leq \epsilon$ and $ c'.n - R.min_n \leq \epsilon$ then Add c' to R ; mark c ; $Growing(c', R)$

Figure 5: The region growing algorithm

The algorithm terminates once all cells have been assigned to regions. Should too many regions be formed, the smallest ones can be dropped and joint to adjacent regions. Centroids of the final regions become the new reference points.

Figure 6b shows a possible outcome of the region growing algorithm on sample data. In my opinion this figure demonstrates the pitfall of the algorithm, i.e. its proneness to reproduce the spatial diversity problem within a local region. In particular, the upper-left region contains a dense hotspot as well as a large sparse area.

I believe that this trap is a result of the rather unfortunate simplifying decision to partition space into Voronoi cells of the identified centroids. More precisely, it is likely that a sparse area is much larger than a neighboring dense area. The Voronoi border between two cells cuts through the midpoint between the two centroids. However, since the dense area is smaller in radius, the optimal border should in fact be closer to its centroid as depicted in Figure 6c.

This analysis suggests that the goal to “discriminate between regions of different densities” (section 4.1) would be fulfilled to a large extent if the space partitioning took into account the relative sizes of neighboring regions as well as their centroids. I expect that a reasonably conformant solution can be found without dramatically increasing the computational overhead.

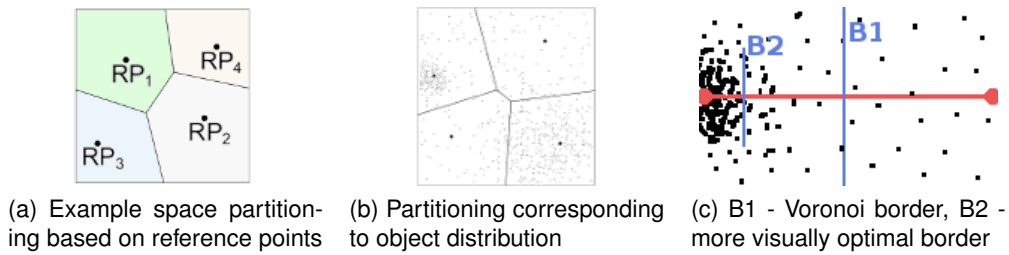


Figure 6: Examples of region growing outcomes

Alternately to region growing, in periodic environments such as city traffic, reference points can be observed and programmed in advance. This approach is expected to reduce the computational overhead at each rollover time, but will make the index less adaptive to change.

As described in section 4, grid granularity in grid-based indexes affects query and update performance. The authors of [1] proved that the optimal compromise between update and query cost is achieved at grid granularity $\lambda = \lceil \frac{1}{2} \log_2 N - \frac{1}{2} \log_2 C_L \rceil$ with N the number of objects and C_L the capacity of leaf nodes.

6 Performance Evaluation

The performance of the ST²B-tree was experimentally compared to the previous solutions: the TPR*-tree and the B^x-tree, both introduced in section 3.

The data used for the experiments were randomly generated objects with speeds in the set $\{30, 60, 90, 150, 300\}$ km/h in a 100×100 km² space. Many experiments use non-uniform object distribution in space such that k randomly selected hotspots give rise to Gaussian-distributed objects around them. Non-uniform queries use the same distribution. Unless mentioned otherwise, the query to update ratio is 1 : 100 in line with a typical real-world MOD environment.

Distribution and changes in object speeds were not documented in [1], so one can not exclude the possibility that the objects were generated with constant speeds, which would be a favorable setup for the ST²B-tree as all predicted locations would fit perfectly.

All tests were run a PC with a Pentium IV 3.0GHz processor, 1GB memory and 80G SATA Disk, running Windows XP. The disk page size was 4KB and a 50-page LRU buffer was used.

Both the ST²B-tree and the B^x-tree use the grid granularity explained in the previous section.

6.1 Spatial Diversity

With a rising number of objects in the database, the update times of the ST²B-tree and the B^x-tree are fairly constant. However, as shown in Figure 7a below, updates on the TPR*-tree not only run considerably slower but also degrade with the number of objects. This is due to high MBR adjustment costs as well as smaller node capacities which result in faster height growing.

The use of a single grid in the B^x-tree causes many false positives that need to be filtered while querying the hotspots. As hoped, the B^x-tree thus performs worse compared to the ST²B-tree (see Figure 7b). In contrast, the TPR*-tree which by design handles hotspots well, suffers from the time overhead associated with MBR enlargement.

Figure 7c shows the increase in kNN query times with rising k . Both the B^x-tree and ST²B-tree evaluate kNN queries as incremental range searches. The initial search window is estimated based on object density. While the ST²B-tree relies on local object density, the B^x-tree only considers the overall object density. Consequently, the initial window is too large in dense regions and too small in sparse ones and more range queries have to be conducted altogether.

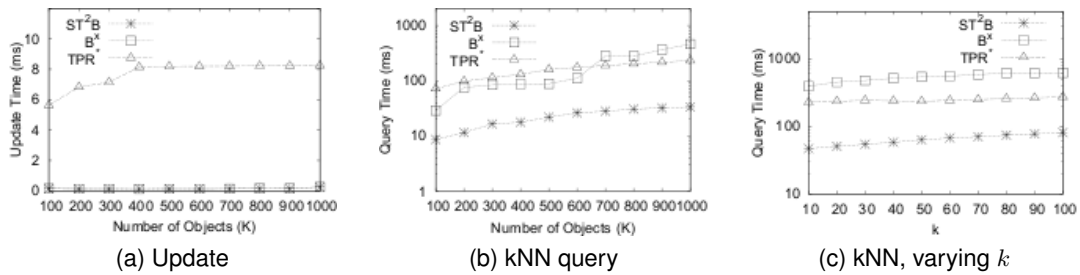


Figure 7: Spatial diversity test results

6.2 Temporal Diversity

The temporal test highlights the benefit of adjusting grid granularities with time. There are $k = 10$ hotspots as before. The experiment consists of two parts, first the number of objects in the system starts small and is systematically increased for 10 rounds. The second part is run in reverse. The benefit of the ST^2B -tree over the B^x -tree is defined as

$$benefit = \frac{\text{total processing time of the } B^x - \text{tree}}{\text{total processing time of the } ST^2B - \text{tree}}$$

The B^x -tree infers the grid granularity based on the initial state of the system. Therefore it starts with a coarse grid and keeps it throughout the experiment. Adding more objects results in overflow pages in the underlying B^+ -tree index and consequent performance degradation on both updates and queries. In comparison, the ST^2B -tree continuously adjusts to the data cardinality changes and thus increasingly gains in benefit over time.

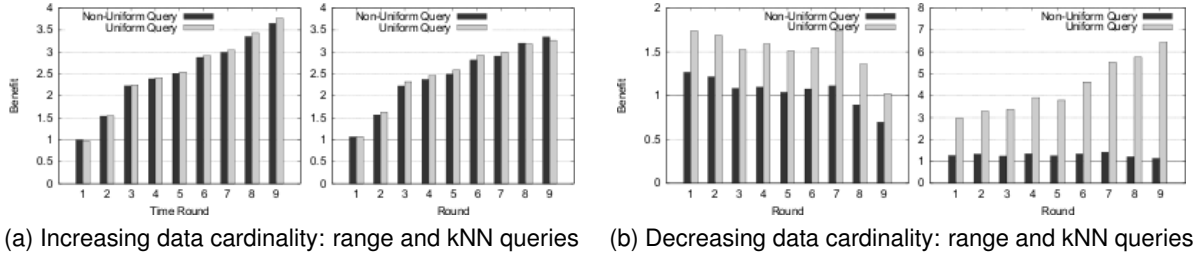


Figure 8: Temporal diversity test results

In the reverse process the B^x -tree uses a fine grid. Similarly, the ST^2B -tree employs a fine grid in the hotspots. On the other hand, in sparse regions the ST^2B -tree uses a more appropriate coarse grid which entails less computational overhead. This explains why in Figure 8b the benefit of uniform queries, which include a comparably higher portion of queries over the sparse regions, is higher than that of the non-uniform ones.

6.3 Spatio-Temporal Diversity

In the spatio-temporal test the number of hotspots in the system is varied with time. Starting with 10 hotspots, their number is reduced by 1 at each round, culminating in just a single Gaussian-distributed hotspot. Samples of data distribution are shown in Figure 9a.

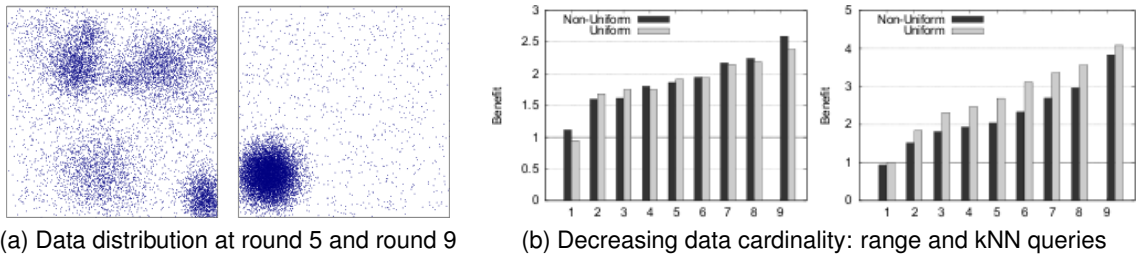


Figure 9: Spatio-temporal diversity test data distribution and results

As expected, the benefit of the ST^2B -tree becomes more significant with increasing data skew. Notably, the performance gain is much higher when it comes to kNN queries.

6.4 Throughput Test

Since the concurrent operation of the TPR*-tree has previously been shown to be inadequate, the throughput test targets the performances of only the two remaining indexes in concurrent environment.

The first experiment with results in Figure 10a runs 8 threads executing varying update/query ratios r . Queries have to examine multiple paths in the tree and consequently execute much slower than updates. Furthermore, although not mutually exclusive, queries prevent the execution of updates while they are running. Throughput is defined as the number of operations completed per second. As updates run faster, the throughput grows with increasing r .

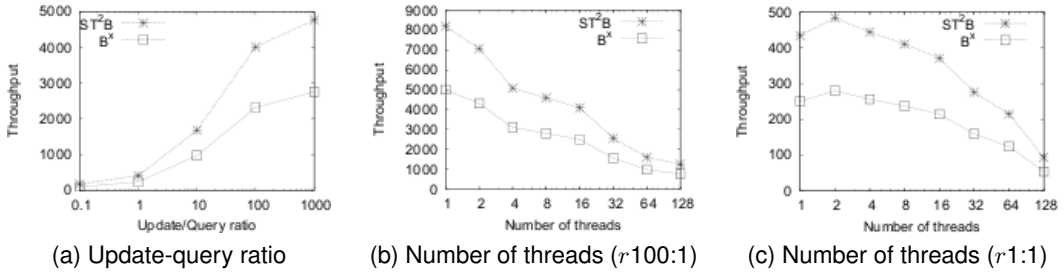


Figure 10: Throughput test results

Figure 10b shows a throughput decay with a rising degree of concurrency. An update both read- and write-locks the node it operates on. As the number of updates exceeds the number of queries, with more threads, more parts of the index are being frequently locked, finally resulting in a drop in throughput.

In my opinion, the necessity of update read-locks is disputable. It might be a good idea to sacrifice some consistency in query outcomes for the sake of significantly faster query evaluation.

Similarly to 10b, a throughput drop can be observed with a 1:1 ratio r in Figure 10c. However, the fact that queries can be run concurrently is reflected in the peak shift. In modest numbers, adding threads in this setting contributes to higher overall throughput.

Notably, the throughput of the ST²B-tree is consistently higher than that achieved by the B^x-tree - owing to the structural and conceptual improvements introduced in [1].

7 Conclusion

The authors of [1] proposed a self-tunable spatio-temporal index for Moving Object Databases. The solution borrows some ideas from its predecessor, the B^x-tree model. By partitioning the space into regions of roughly constant density and assigning a tailor-made grid to each of them, the model gains the ability to accommodate for data diversity in space as described in section 4. Furthermore, the index periodically revises the regions, thus keeping up to date with the data distribution throughout time. Consequently, the temporal and spatio-temporal diversities are taken into account as well.

Although the tuning procedure that runs at every transition time comes with a time penalty, experiments discussed in section 6 show that the ST²B-tree consistently outperforms its two main competitors when dealing with uneven and/or time-varying distributions on data.

In summary, the resulting ST²B-tree index to a large extent achieved the goals listed in section 4.1. Moreover, it is an efficient, robust and scalable structure. To conclude, the ST²B-tree combines the adaptability of the TPR*-tree with the efficiency of the B^x-tree and as a result is in certain situations superior to both of these.

References

- [1] S. Chen, B.C. Ooi, K.L. Tan, and M.A. Nascimento. ST²B-tree: a self-tunable spatio-temporal B+-tree index for moving objects. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 29–42. ACM New York, NY, USA, 2008.
- [2] C.S. Jensen, D. Lin, and B.C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, page 779. VLDB Endowment, 2004.
- [3] R. Ramakrishnan and J. Gehrke. Database management systems, 3rd edition, 2003.
- [4] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, page 801. VLDB Endowment, 2003.
- [5] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, volume 175, 1998.