International Journal of Control Theory and Applications

ISSN : 0974-5572

© International Science Press

Volume 10 • Number 11 • 2017

Comparative Analysis of BB^x, OBB^x, POBB^x and SOBB^x Indexing Methods of Spatial Databases

K. Appathurai[1] and M. Anandkumar[2]

[1] *Assistant Professor Dept. of Information System and Technology, Sur University College, Sur, Oman*
[2] *Associate Professor Dept. of Information Technology, Karpagam University, Coimbatore – 21*

Abstract: Spatiotemporal database deals with moving objects that change their locations over time. Generally, moving objects report their locations obtained via location-aware devices to a spatiotemporal database server. The server can store all updates from the moving objects so that it will be capable of answering queries about the past. Some applications need to know current locations of moving objects only. In this case, the server may only store the present status of the moving objects. To predict future positions of moving objects, the spatiotemporal database server may need to store additional information, viz., the objects' velocities. In this study the indexing structures BB^x, Optimal BB^x, Parameterized OBB^x and Space based OBB^x are analyzed in the way of tree creation, methodology of indexing, the process of updation and the process of migration are studied.

Keywords: Moving Objects, BB^x index, OBB^x index, POBB^x index, SOBB^x index and Migration.

I. INTRODUCTION

In real world applications spatiotemporal databases store data which are continuously varying in space and time. Because it produce a huge volume of data compared to the traditional database applications. Consequently they need to be managed efficiently, in order to process the information in a sensible manner. Additionally, unlike traditional DBMS operations, in spatiotemporal operations, both the cost of I/O, as well as the cost of computation is quite high. Thus efficient storage and indexing techniques are very much important in handling and processing this kind of data (3,4,5).

Nowadays there is a tremendous development in the statistical models and techniques to analyze spatiotemporal data such as vehicle detection and monitoring data (11). Spatiotemporal data occur in many other contexts e.g. disease mapping and economic monitoring of real estate prices (1). Besides the key interests in analyzing such data are to produce smooth and predict time evolution of some response variables over a certain spatial domain. Naturally, such predictions are made from data observed on a large number of variables which themselves vary over time and space. There are many other significant areas where spatiotemporal data are used to detect familiar and meaningful patterns as well as to make predictions (12-15). Examples consist of hydrology, ecology, geology, social sciences and many areas in medicine such as brain imaging, wildlife population monitoring and tracking, and machine vision (1).

The ultimate goal of indexing the moving object data is to speed up the retrieval operations of a database table. But the performance of indexing is hindered due to the following reasons.

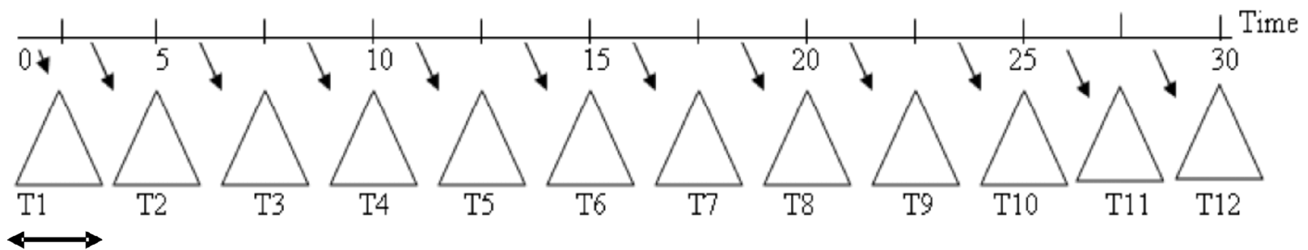
1. During indexing (2) the moving objects can't be updated within the maximum update interval should be migrated. Due to more migration process the performance is not worthy. So data at all points of time is constructed with empirical experiments and are reported. So one such case occurs when half objects are updated frequently while half are not, resulting in relatively many forced updates (i.e. updation by the system).
2. Usually after the lifespan of tree the object data is moved from one tree to another by updation or migration. During updation or migration the object is removed from old tree and then added to the next tree. Every object has indexed time based on that it will find out the old tree and then finds out the position of the object in that tree and after that the object is removed. Here for finding the old tree and the position the searching took more time.
3. As per cost and density it may have the following problems in some scenarios:
    1. Larger tree leads to an increase in searching cost, while a smaller tree may introduce extra migration cost.
    2. Both the updation time and migration time becomes larger in case of high density (population of objects) in a tree.

II. BROAD BINARY INDEX (BB^x INDEX)

2.1. Tree Construction

Figure 1 shows the tree creation of BB^x index structure (2). In the below figure T1, T2, etc., are trees. In this example consider the number of moving objects is 50 and found the maximum update time interval among all the 50 objects (maximum time interval means the frequency of update time interval to the server), say in this case it is 5 seconds means, based on this the linear representation is formed as shown in the figure 1. In this case 12 trees are used for the time period 0 to 30 seconds. Initially the trees are formed 5, 10, 15 etc., seconds and for storage efficiency the interval is divided by n (usually n=2) (Dan Lin, 2005) so for the seconds 2.5, 7.5, 12.5 etc., seconds also the trees are formed. Here the interval is divided by n i.e. 5/2 = 2.5 (subintervals).

The Maximum Update time interval is 5 sec.



This arrow indicates the lifespan of tree T1.

Figure 1: Tree creation of BB^x index structure

2.2. Indexing Method

Each tree has a lifespan (a minimum and maximum time period for indexing). At the end of the lifespan the tree values are updated to the next tree. So first, it is checked whether all the objects reach the next tree or not and they reach then all the objects to next tree are updated and then the objects are removed or deleted from the

existing old tree to avoid duplication of index. The function updation is called for updation of objects. In this updation algorithm the tree where the update object is identified (search) and located and then the position of the object in that tree is found out (search). Then the identified object is removed and updated in new tree. The object insertion or updation into the tree the binary tree method (2) is followed, If the key value of insertion node 'C' is lesser than the key value of node 'N' then node 'C' is inserted on left position of the node 'N' and if it is greater, it is inserted on right position of the node 'N'. If already nodes are there in that position it assumes that node as 'N' and the same procedure is followed. The insertion time for each object is stored in 'Arr' and total object inserted is stored in 'Tot'.

2.3. Updation and Migration Technique

Each tree has a lifespan. At the end of the lifespan the tree values are updated to the next tree. So first, it is checked whether all the objects reach the next tree or not and if they reach then all these objects are updated to next tree and then those objects are removed or deleted from the existing old tree to avoid duplication of index. For updation of objects the function updation is called. If some of the objects do not reach the next tree, for those objects the migration function is called. In this case the property of velocity is found missing because of non-moving object. So for those migrated objects the future position can't be predicted. The following steps show the Migration Algorithm.

III. OPTIMAL BROAD BINARY INDEX (OBB^X INDEX)

3.1. Tree Construction

In the proposed work i.e. Optimal Broad Binary Index (OBB^x), the maximum update time interval is doubled. The maximum update interval value 5 is doubled (i.e. 10) and the linear representation is formed as shown in the figure 2. Initially the trees are formed 10, 20, 30 seconds and for storage efficiency the interval is divided by n (usually n=2) so for the seconds 5, 15, 25 seconds also the trees are formed. Here the interval is divided by n i.e. 10/2 = 5 (subintervals).

The Maximum Update time interval is 10 sec.

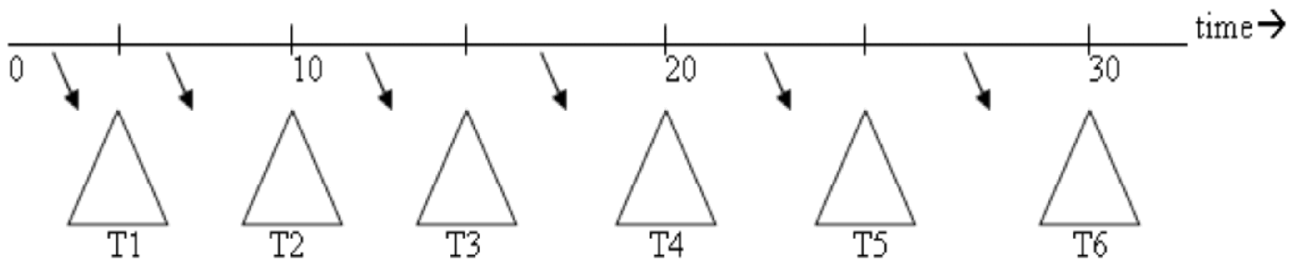


Figure 2: The OBB^x index tree

3.2. Indexing Method

The following algorithm shows for tree construction, Object insertion, updation and migration.

1. The maximum update time interval for each object is found out and stored in 'ui'.
2. The maximum update time interval 'ui' is multiplied by two and then based on this interval the linear array is formed for t_{s1}, t_{s2}, t_{s3}, etc.
3. Array of 'n' equal intervals of t_{s1}, t_{s2}, t_{s3}, etc., is formed (subintervals).

4. Each object lifespan is found out and is stored in 'LE'.
5. Based on the lifespan the data are stored in the tree.
6. If the key value of insertion node 'C' is lesser than the key value of node 'N' then node 'C' is inserted on left and if it is greater, it is inserted on right. If already nodes are there the same procedure is followed. The insertion time for each object is stored in 'Arr' and total object inserted is stored in 'Tot'.
7. After the lifespan, the objects move from one tree to another, while 'Arr' is not equal to null, and it is checked whether all the moving objects are reach the new tree or not. If they reached function is called as update or else function is called as migration.

Each tree has a lifespan (a minimum and maximum time period for indexing). At the end of the lifespan the tree values are updated to the next tree. So first, it is checked whether all the objects reach the next tree or not and they reach then all the objects to next tree are updated and then the objects are removed or deleted from the existing old tree to avoid duplication of index. The function updation is called for updation of objects. The following algorithm shows how the updation takes place in OBB^x index. In this updation algorithm the tree where the update object is identified (search) and located and then the position of the object in that tree is found out (search). Then the identified object is removed and updated in new tree. The object insertion or updation into the tree the binary tree method (Dan Lin, 2005) is followed, If the key value of insertion node 'C' is lesser than the key value of node 'N' then node 'C' is inserted on left position of the node 'N' and if it is greater, it is inserted on right position of the node 'N'. If already nodes are there in that position it assumes that node as 'N' and the same procedure is followed. The insertion time for each object is stored in 'Arr' and total object inserted is stored in 'Tot'. The figure 3 shows the overall structure of indexing.

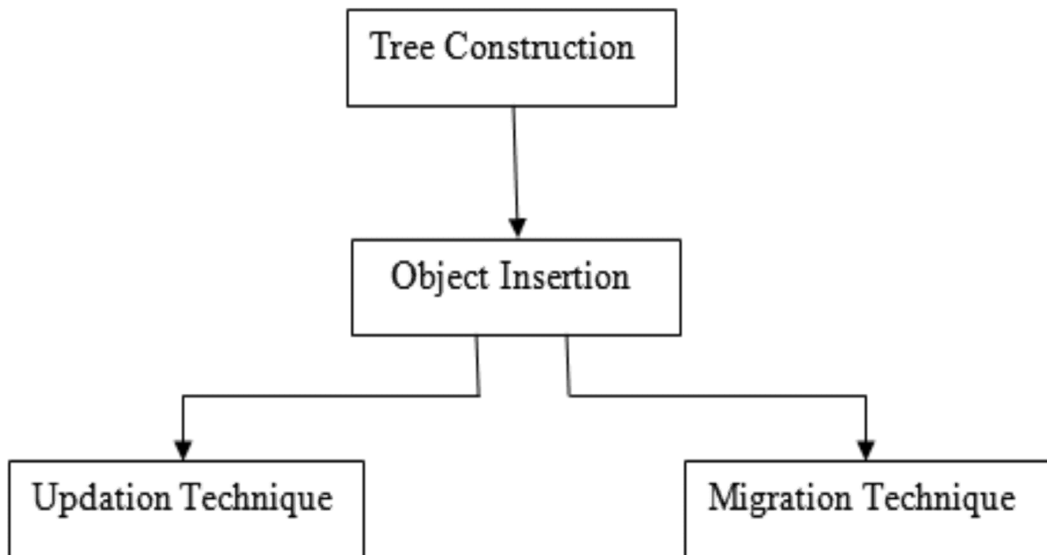


Figure 3: Overall Structure of Indexing

3.3. Updation and Migration Technique

The following steps show the updation algorithm. In this 'tindex', 'posindex' and 'keyo' are the variables for storing the values of old objects index time in the tree, position of the object within the tree and key value of the object. Here the objects previous tree value is found using 'tindex' value.

1. Here Eo and En are old and new objects respectively Input:

   tindex ⟵ time Eo is indexed in the tree

   find (search) tree Tx whose lifespan contain tindex

2. Find the position of the object in the tree

   posindex ⟵ position of Eo at tindex

3. Locate Eo in Tx according to keyo

   keyo ⟵ x-value of the posindex

4. Modify the end time of Eo's lifespan to current time (removal)

   t'index ⟵ time En will be indexed

   pos'index ⟵ position of En at t'index

   keyn ⟵ x-value of the pos'index

5. Insert En into the latest tree according to keyn

Each tree has a lifespan. At the end of the lifespan the tree values are updated to the next tree. So first, it is checked whether all the objects reach the next tree or not and if they reach then all these objects are updated to next tree and then those objects are removed or deleted from the existing old tree to avoid duplication of index. For updation of objects the function updation is called. If some of the objects do not reach the next tree, for those objects the migration function is called. In this case the property of velocity is found missing because of non-moving object. So for those migrated objects the future position can't be predicted. The following steps show the Migration Algorithm.

In the above algorithm, tindex is a variable which is used to store the time when the object was indexed into the old tree. Posindex is another variable which is used to store the position of the object in the old tree. 'Keyo' is used to store the key value of the object. The variable 'curpos', 'curtim' and 'curarr' are variables used to store the new object values.

The Optimal Broad Binary (OBB^x) indexing is calculated using formulas 1, 2 and 3 mentioned below. In the below formula $[phase]_2$ and $[x_rep]_2$ like the format $[x]_2$, it refers the binary representation of x, and $\oplus$ denotes concatenation. The two components of the function are 'phase' and 'x_rep' which are defined in formulas 2 and 3. Here 'O' is a given object and t_u is the time when the object issues an update.

IV. PARAMETERIZED OPTIMAL BROAD BINARY INDEX ($POBB^X$ INDEX)

4.1. Tree Construction

In case of $POBB^x$ indexing the tree construction is same as OBB^x indexing method. i.e the maximum update time interval is make it as twice, based on that the linear representation is formed.

1. Here Eo and En are old and new objects respectively

   find (search) tree Tx whose lifespan contain tindex

   tindex ⟵ time Eo is indexed in the tree

2. Based on tindex the position of the object is find out

   posindex ⟵ position of Eo at tindex

3. Locate Eo in Tx according to keyo

   keyo ⟵ x-value of the posindex

4. Modify the end time of Eo's lifespan to current time

5. Curpos ⟵ Current position of the object En

6. Curtim ⟵ Current time of the object En

7. Curarr ⟵ Current tree of the object En which lifespan contains Curtim

4.2. Indexing Method

A proposed indexing algorithm called Parameterized Optimal Broad Binary (POBB^x) index is implemented from the OBB^x index algorithm. In OBB^x index structure the searching process is one of the major crises during updation and migration processes. Besides the searching takes more time during the updation and migration processes. The workload of whole process of indexing also required more effort. Owing to these efforts, there was a great increase in the memory space utilization, processor utilization, execution time and cost. Moreover in the tree the node insertion and deletion is also a complex processes when the number of moving objects is high.

The main aim of the proposed algorithm is to reduce the searching process during updation and migration of moving objects, so that the efficiency is improved and we get better results than in the OBB^x index algorithm.

4.3. Updation and Migration Technique

In OBB^x index algorithm during node value updation or migration first it finds the old tree by taking the lifespan using updated time against it in the old tree. Next it finds the position of the node in that tree and then it finds the key value of the node. There the end time is changed to current time. So for each updation or migration, searching plays a major role. In this proposed work the maximum update time interval is similar to OBB^x index and during values transferred from one tree to another the old tree address and position is also passed along with the moving object. So during updation or migration from one tree to another there is no need to search for the old tree and its position. So lot of searching time and effort is reduced. Due to this reduction of searching the node access is also reduced. Besides, the utilization of memory is also reduced and automatically the processing speed is improved than in OBB^x index. The algorithm to Tree Construction, Object Insertion, Updation and Migration were similar to OBB^x index method but the searching technique is differing from OBB^x index algorithm.

V. SPACE BASED OPTIMAL BROAD BINARY INDEX (SOBB^X INDEX)

5.1. Tree Construction

In case of SOB**B^x** indexing the tree construction is same as OB**B^x** indexing method. i.e the maximum update time interval is make it as twice, based on that the linear representation is formed.

5.2. Indexing Method

The main aim of the proposed Space based Optimal Broad Binary (SOBB^x) algorithm is to improve the performance in exclusive level than in POBB^x indexing technique. In this proposed algorithm, the basic design of POBB^x indexing structure bears the major work focus in node updation in the trees. The new technique applied is called Hybrid update. In Moving object indexing, the Location of Objects, Distribution of Objects, and Workload are the three factors play an important role for effective indexing and they can change frequently based on time. In order to avoid migration as much as possible while keeping the tree size relatively small, we have applied Hybrid Update technique in POBB^x indexing.

5.3. Updation and Migration Technique

The principle of Hybrid update is to update as many objects as possible without increasing the number of input/ output accesses. This means, the object identity, current location and velocity for each moving object are recognized. Based on this information, the future is predicted and Hybrid update is applied. This hybrid update accesses the same tree nodes as regular updates. In a Time t, more objects are shifted from a current tree to some distanced tree instead of the next immediate tree. Fewer objects are left in the older sub tree and it may reduce the migration process. This saves the cost of regular update as well. So we can effectively index the moving objects by nearly 20-25% more efficiently than by POBB^x indexing method. Updation costs and migration costs are reduced upto 20-30% when compared to POBB^x indexing method. Figure 4 shows the Space based Optimal BB^x index Hybrid Update. It clearly shows how the tree is constructed based on time interval and the Hybrid Update. Besides it indicates some of the objects updated from tree T1 to T4, T1 to T8 instead of regular update.

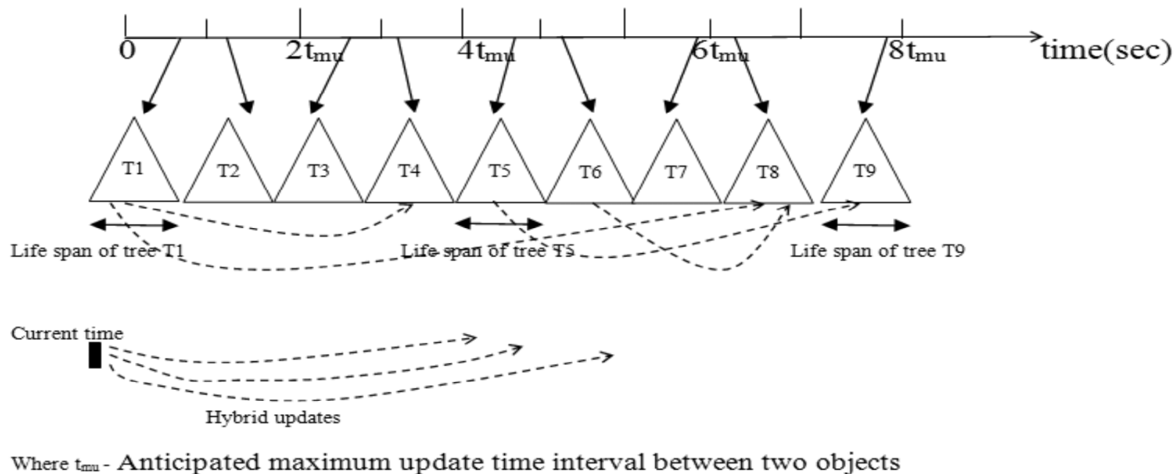


Figure 4 : The SOBB^x index Hybrid Update

VI. PERFORMANCE ANALYSIS

In this indexing nine moving objects are considered. The starting time of indexing is 13 ms and the ending time is 201 ms. In figure 5 the 'x' axis is the indexing methods and 'y' axis is the indexing time.

Figure 5 shows the total indexing time for all the four methods viz., Broad Binary index (BB^x index), Optimal Broad Binary index (OBB^x index), Parameterized Optimal Broad Binary index (POBB^x index) and Space based Optimal Broad Binary index (SOBB^x index). The total processing time for BB^x Indexing is 17 sec., the total processing time for OBB^x Indexing is 10.1 sec., the total processing time for POBB^x Indexing is 8.6 sec., and the total processing time for SOBB^x index is 6.5 seconds. So it is proved that the SOBB^x index method is

much better than BB^x, OBB^x and $POBB^x$ methods. The percentage of improvement is calculated by the following formula,

$$(x1-x2) / x1 * 100.$$

In this 'x1' is the first value and 'x2' is the second value.

Besides in figure 5, the percentage of improvement from BB^x index to OBB^x index is 40.5%, from OBB^x index to $POBB^x$ index is 14.85% and from $POBB^x$ index to $SOBB^x$ index is 24.4%. The overall improvement from BB^x index to $SOBB^x$ index is 62%. The percentage of improvement is calculated as (17- 6.5) / 17 * 100.

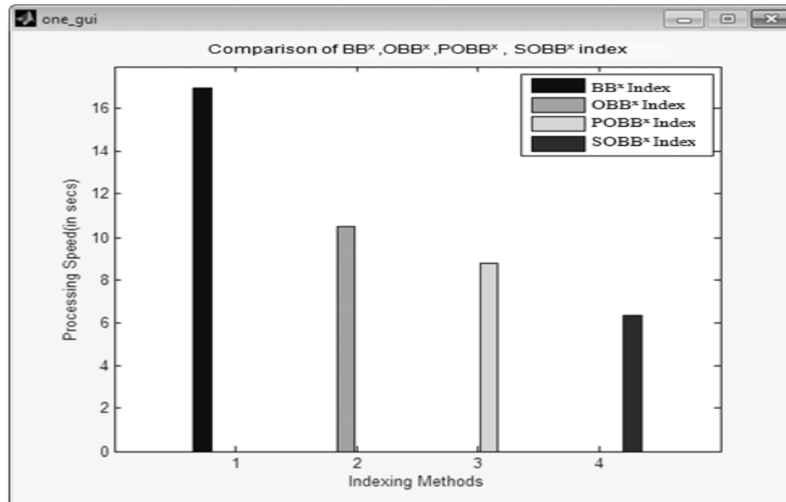


Figure 5: Comparison of $BB^{x,}$ OBB^x, $POBB^x$ and $SOBB^x$ indexing in terms of Processing Time

Figure 6 indicates the number of trees used in indexing process for all the four techniques. Here the number of trees used in BB^x index is 25. The number of trees used in OBB^x index, $POBB^x$ index and $SOBB^x$ index is 13. This is because of the maximum update time interval. In case of OBB^x index, $POBB^x$ index and $SOBB^x$ index methods the maximum update time interval is doubled, and so the number of trees used is reduced to almost 50%.

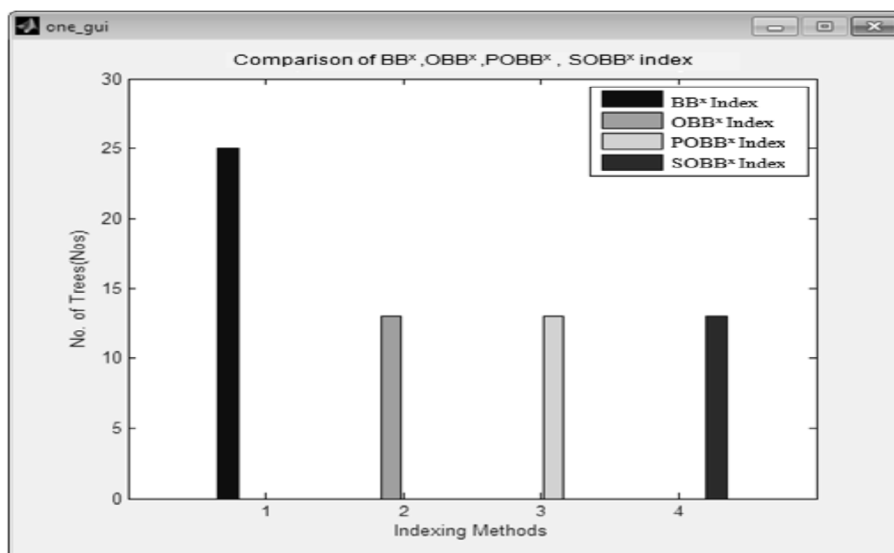


Figure 6: Comparison between BB^x, OBB^x, $POBB^x$ and $SOBB^x$ in terms of creation of number of trees

Figure 7 indicates the number of migration hits that occurred in all the four techniques. Here the number of migration hits in BB^x index is 99. The number of migration hits in OBB^x index, $POBB^x$ index and $SOBB^x$ index is 49. This is because of maximum update time interval. In case of OBB^x index, $POBB^x$ index and $SOBB^x$ index methods, the maximum update time interval is doubled, and so the number of migration hits is reduced to almost 50%. Once the migration is reduced the efficiency is automatically improved.

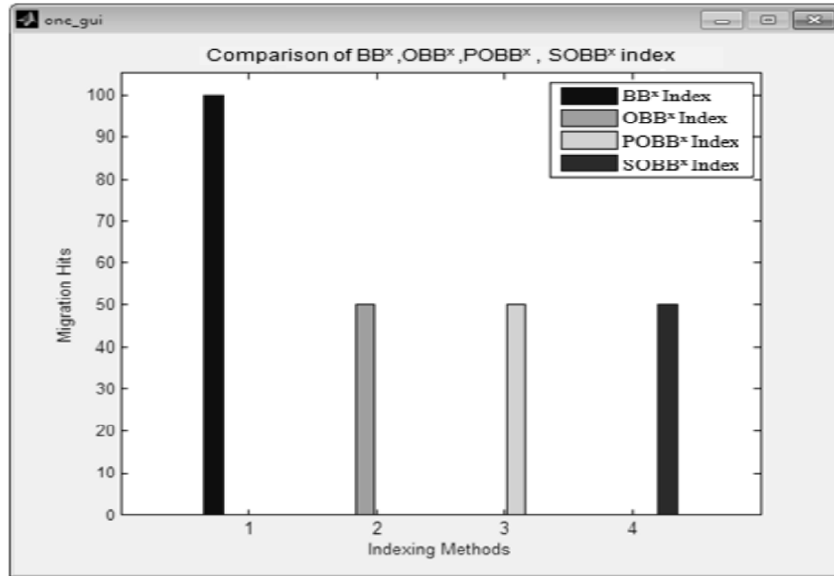


Figure 7: Comparison of BB^x, OBB^x, $POBB^x$ and $SOBB^x$ indexing in terms of migration hits

Figure 8 indicates the number of node accesses occurred in all the four techniques. Here the number of nodes used in BB^x index is 720. The number of nodes used in OBB^x index is 645, in $POBB^x$ index is 528 and in $SOBB^x$ index is 504. This is because of the new searching mechanism which is implemented in $POBB^x$ index and by hybrid update technique in $SOBB^x$ index. So once the node usage decreases, automatically it will reduce the work load and improve the efficiency.

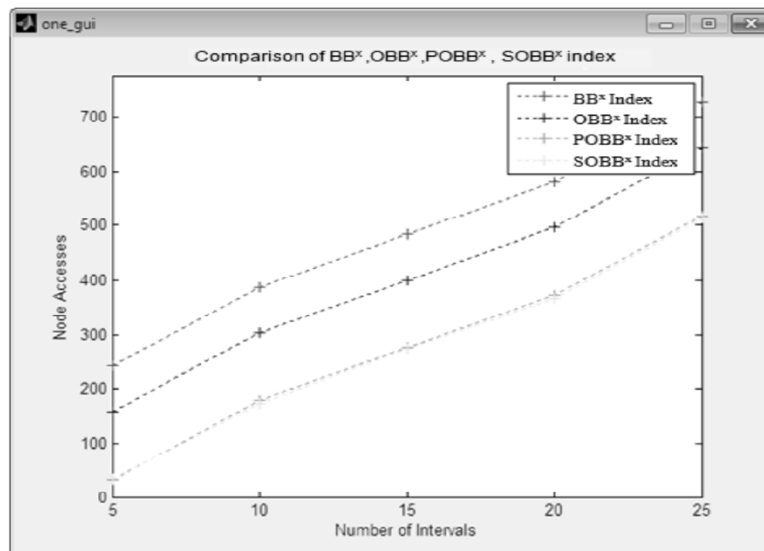


Figure 8: Comparison of BB^x, OBB^x, $POBB^x$ and $SOBB^x$ indexing in terms of Node Accesses

Figure 9 indicates the usage of memory in all the four techniques. Here the memory space required in BB^x index is 98 bytes. The memory space required in OBB^x index is 38 bytes. The memory space required in $POBB^x$ index is 32 bytes and the memory space required in $SOBB^x$ is 29 bytes. This is because of the new searching mechanism which is implemented in $POBB^x$ index and by hybrid update technique in $SOBB^x$ index. So once the memory usage decreases, automatically it will reduce the work load, cost and the efficiency are improved.

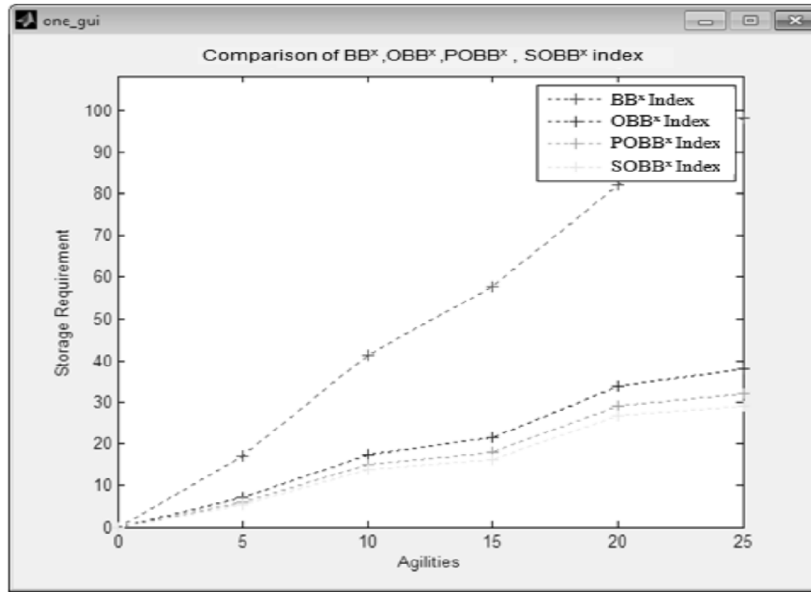


Figure 9: Comparison of BB^x, OBB^x, $POBB^x$ and $SOBB^x$ indexing in terms of Storage

Figure 10 shows the updation time for all the four indexing methods. Here the updation time for BB^x index is 8.4 seconds. The updation time for OBB^x index is 6.4 seconds. The updation time for $POBB^x$ index is 5.0 seconds and for $SOBB^x$ is 2.2 seconds. Here the updation time drastically decreases from $POBB^x$ index method to $SOBB^x$ index method because of the hybrid update technique in $SOBB^x$ index. The percentage of updation time improvement from $POBB^x$ index to $SOBB^x$ index is almost 56%.

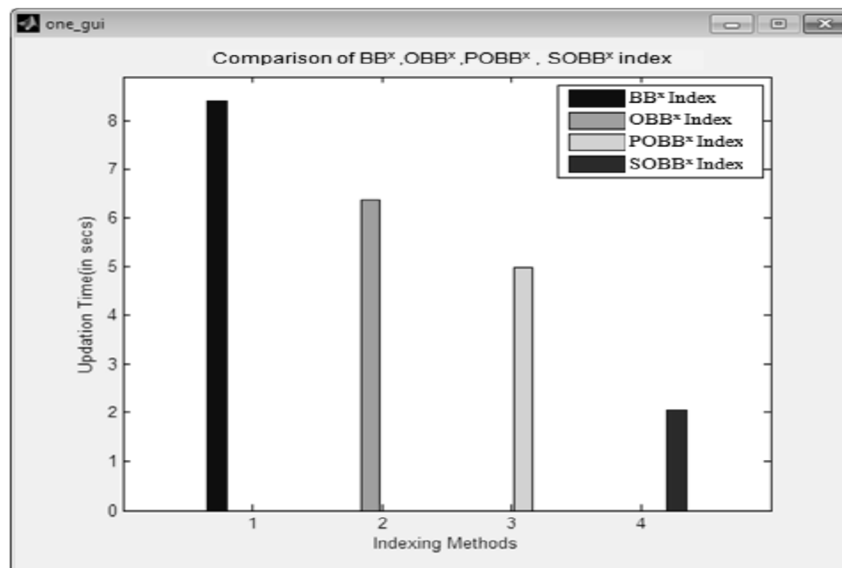


Figure 10: Total Updation Time

Figure 11 shows the migration time for all the four indexing methods. Here the migration time for BB^x index is 1.5 seconds. The migration time for OBB^x index is 1.1 seconds. The migration time for POBB^x index is 0.99 seconds and for SOBB^x is 0.76 seconds. Here the migration time gradually decreases from BB^x index to SOBBx index method because of the new searching, hybrid update technique in POBB^x index and SOBBx index.

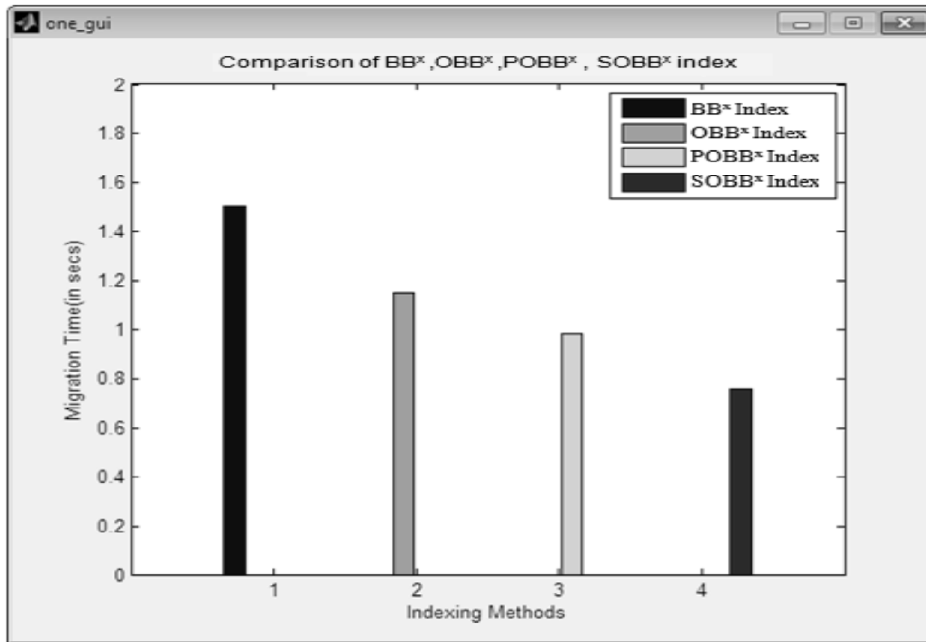


Figure 11: Total Migration Time

5.8. Results and Discussion

- The number of Moving Objects is considered to be : 9
- Starting Time: 13 ms.
- Ending Time: 201 ms.

Table 1
Comparison of all the four indexing methods

Aspects	*BB^x*	*OBB^x*	*POBB^x*	*SOBB^x*
Processing Time (sec.)	17	10.1	8.6	6.5
No. of Trees	25	13	13	13
Migration Hits	99	49	49	49
Node Accesses	720	645	528	504
Storage Requirement (bytes)	98	38	32	29
Updation Time (sec.)	8.4	6.4	5.0	2.2
Migration Time (sec.)	1.5	1.1	0.99	0.76

Table 1 shows the comparison results of BB^x index, OBB^x index, POBB^x index and SOBB^x index methods under different aspects. Table 2 shows the comparison in terms of processing time.

Table 2
Comparison based on processing time

No. of Objects	*BB^x*	*OBB^x*	*POBB^x*	*SOBB^x*
9	17	10.1	8.6	6.5
% of improvement (Processing Time)				

The above comparisons clearly show that the indexing performance is improved almost to 62% from BB^x index to $SOBB^x$ index method. Moreover the performance analysis is conducted at different number of moving objects for all the four indexing methods in terms of indexing time. Table 3 shows the results of all the four methods.

Table 3
Performance analysis

S. No.	*No. of Objects*	*BB^x*	*OBB^x*	*POBB^x*	*SOBB^x*
1	50	79	50	42	31
2	100	155	89	75	58
3	150	232	135	117	88
4	200	309	184	155	118
5	250	385	235	198	146
Avg. Processing time in Sec.		232	138.6	117.4	88.2

The indexing is done at different number of moving objects like 50,100,150,200 and 250. In all the cases the percentage of improvement is almost the same. Finally the average processing time is calculated for all the methods and the percentage of improvement from BB^x index to OBB^x index is found to be around 40%, from OBB^x index to $POBB^x$ index is around 15%, from $POBB^x$ index to $SOBB^x$ index is around 25% and the overall improvement from BB^x index to $SOBB^x$ index is around 62%.

ACKNOWLEDGEMENT

I sincerely thank Sur University College for providing us with various resources and an unconditional support for carrying out this work.

VII. CONCLUSION

The first contribution of the paper is to minimize the number of trees used in the indexing methods. Normally in the indexing methods more number of trees is involved leading to higher level of workload, indexing time and migration hit ratio. While minimizing the number of trees involved in indexing the migration hit ratio, indexing time and the workload automatically get minimized and as a result the efficiency of the proposed algorithm will be great impact. With the proposed work, the number of trees and the migration process is found to be reduced to almost 50%. The second contribution of the paper is to minimize the searching time during indexing process. Usually in index structure the searching process will be one of the major crises during moving object updation and migration processes. Normally, the searching takes more time during the moving object updation and migration processes and the workload of whole process of indexing will require more effort. Owing to these efforts, there becomes a great increase in the memory space utilization, processor utilization, execution time and cost. So in this proposed work a new technique is implemented to reduce the searching time, memory space utilization, processor utilization, execution time and cost.

The Third contribution of the paper is further enhancement of indexing based on the cost and density (population) of the moving objects. In some scenarios, larger tree lead to a higher query cost, while a smaller tree may introduce extra migration cost. Besides, both the updating time and migration time becomes more in case of high density in a tree. In the proposed work, in order to avoid migration at the most while keeping the tree size relatively small, a new update technique is applied. In the present investigation, it is illustrated that the past indexing algorithms are not suitable for modern day applications, due to the advances in the indexing technologies and searching mechanisms. The primary contribution is to overcome the weakness mentioned above. The proposed research is carefully designed in such a way that it can be used efficiently in monitoring with high performance. It is well designed so as to cope up with the real time applications. The simulation results show that the performance is good when compared with the other indexing algorithms.

REFERENCES

[1] Guoliang Xing, Jianping Wang, Ke Shen, Qingfeng Huang, Xiaohua Jia and Hing Cheung So, 2008. Mobility-assisted Spatiotemporal Detection in Wireless Sensor Networks. IEEE conference on 1063-6927.

[2] Dan Lin, Christian S. Jensen, Beng Chin Ooi, Simonas S¡ altenis, 2005. BB^x index Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects. ACM Ayia Napa Cyprus, 1- 59593 – 041.

[3] Christian S. Jensen_, Dan Lin, and Beng Chin Ooi, 2004. Query and Update Efficient B+ -Tree Based Indexing of Moving Objects, Proceedings of the 30th VLDB Conference, Toronto, Canada, 768-779.

[4] Chengcui Zhang, and Shu-Ching Chen, 2003. Adaptive Background Learning for Vehicle Detection and SpatioTemporal Tracking, IEEE conference on ICICS-PCM, Singapore, 0-7803-8185.

[5] Mete Celik, and Shashi Shekhar, 2006. Mixed-Drove Spatio-Temporal Co-occurrence Pattern Mining: A Summary of Results, Proceedings of the Sixth International IEEE Conference on Data Mining, 0-7695-2701.

[6] Maria Luisa Damiani, Herve Martin, Yucel Saygin, Maria Rita Spada and Cedric Ulmer, 2009. Spatiotemporal Access Control: Challenges and Applications on ACM 978-1-60558-537.

[7] Nasrin, Salma, Kawagoe and Kyoji , 2010. A novel index structure for efficient data management in super-peer architecture, IEEE Conference on, Ubiquitous and Future Networks (ICUFN).

[8] Manachai Toahchoodee, Indrakshi Ray, Kyriakos Anastasakis, Geri Georg and Behzad Bordbar, 2009. Ensuring Spatiotemporal Access Control for Real-World Applications, on ACM 978-1-60558.

[9] Y. Tao, D. Papadias, and J. Sun, 2003. The TPR*-Tree: An Optimized Spatiotemporal Access Method for Predictive Queries. In Proc. of the Intl. Conf. on Very Large Data Bases, VLDB.

[10] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled, 2002. STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. In Proc. of the Workshop on Alg. Eng. and Experimentation, ALENEX, pages 178–193.

[11] George Kollios, Vassilis J. Tsotras, Dimitrios Gunopulos, Alex Delis and Marios Hadjieleftheriou, 2001. Indexing animated objects using spatiotemporal access methods, IEEE Transactions on Knowledge and Data Engineering.

[12] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, 2000. Indexing the Positions of Continuously Moving Objects. In Proc. of the ACM Intl. Conf on Management of Data, SIGMOD, pages 331–342.

[13] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, 1996. Spatiotemporal Indexing for Large Multimedia Applications. In Proc. of the IEEE Conference on Multimedia Computing and Systems, ICMCS.

[14] Mindaugas Pelanis, Simonas Saltenis And Christian S. Jensen, 2005. Indexing the Past, Present and Anticipated Future Positions of Moving Objects, ACM Transactions on Database Systems, Vol. , No. , 05, Pages 1–43.

[15] Su chen, beng chin ooi and kian-lee tan, 2013. Continuous Online Index Tuning in Moving Object Databases, ACM Transactions on Database Systems, Vol. 1, No. 7, Pages 1–45.