A Survey on Spatio-Temporal Access Methods

K. Appathurai
Asst.Prof. And Head
Department of Information Technology
Karpagam University, Coimbatore – 21

Dr. S. Karthikeyan
Director, School of computer Science
Karpagam University
Coimbatore – 21

ABSTRACT
Nowadays many new indexing structures are introduced for spatiotemporal access metods. Spatiotemporal access methods are classified into four categories: (1) Indexing the past data, (2) Indexing the current data, (3) Indexing the future data, and (4) Indexing data at all points of time. In this short survey we consider the 4th category, in that survey mainly deals with the comparative analysis of BBx – index, PCFI+ index and RPPF – TREE

Keywords
Index, query, access methods and insertion method.

1. INTRODUCTION
Moving objects are pact with spatial databases over change their place over time. Generally, moving objects account their places obtained via place-aware devices to a spatio-temporal database server. The server store updates from the moving objects and it is capable of answering queries about the past. Some applications need to know current places of moving objects only. This case, the server may only store the current status of the moving objects. To predict future positions of moving objects, the spatio-temporal database server may need to store additional information, e.g., the objects' velocities [8].

Numerous query types are maintained by a spatio-temporal database server, e.g., range queries "Find all objects that intersect a certain spatial range during a given time interval", x-nearest neighbor queries "Find x restaurants that are closest to a given moving point", or trajectory queries "Find the trajectory of a given object for the past hour". These queries may execute on past, current, or future time data. A large number of spatio-temporal index structures has been proposed to support spatio-temporal queries efficiently [12,13].

This survey is based on [1], where they cover and classify spatio-temporal access methods published in the years 2003 to 2010. Since 2003, new spatial-temporal access methods have been developed that use entirely new approaches or that address weaknesses in existing approaches. Besides having separate indexing structures for past data, present data, or future data, some access methods have been proposed to deal with data at all points in time.

Also having separate indexing structures for past data, present data, or future data, some access methods have been proposed to deal with data at all points in time. Several spatio-temporal indexing methods are proposed for special environments, e.g., road networks or indoor networks

2. THE ACCESS STRUCTURE
2.1 BBx – index
2.1.1 Structure
The BBx-index consists of nodes that in turn consist of entries, each of which is of the form hx rep; tstart; tend; pointeri. For leaf nodes, pointer points to the objects with the corresponding x rep, where x rep is obtained from the space-filling curve; tstart denotes
the time when the object was inserted into the database (corresponding to the tu in the description of the Bx-tree), and tend denotes the time that the position was deleted, updated, or migrated (migration refers to the update of a position done by the system). For non-leaf nodes, pointer points to a (child) node at the next level of the index: tstart and tend are the minimum and maximum tstart and tend values of all the entries in the child node, respectively. In addition, each node contains a pointer to its right sibling to make possible query processing. Let us illustrate the BBx-index with an example. A BBx-index with n = 2. Objects inserted between timestamps 0 and 0:5tmu are stored in tree T1 with their positions as of time 0:5tmu; those inserted between timestamp 0:5tmu and tmu are stored in tree T2 with their positions as of time tmu; and so on. Each tree has a maximum lifespan: T1's lifespan is from 0 to 1:5tmu because objects are inserted starting at timestamp 0 and because those inserted at timestamp 0:5tmu may be alive throughout the maximum update interval tmu, which is thus until 1:5tmu; the same applies to the other trees.

2.1.2 Insertion Algorithm
Insertion into the BBx-index is similar to insertion into the Bx-tree [7,11]. i.e in Bx-tree The insertion algorithm is straightforward. Given a new object, we calculate its index key according to the below euuation,

Bx Value(O, tu) = [index_partition]2 * [x_rep]2

then insert it into the B-tree as in the Bx-tree. To delete an object, we assume that the positional information for the object used at its last insertion and the last insertion time are known. Then we calculate its index key and employ the same deletion algorithm as in the Bx-tree. Therefore, the Bx-tree directly inherits the good properties of the B-tree, and we expect efficient update performance.

To delete an object, we first find the tree where this object is stored. Rather than physically removing the object, we modify the end time of its lifespan, tend, to be the current time.

2.2 PCFI+– index

2.2.1 Structure
The PCFI+-index consists of two parts: one situated in memory, the other one situated on disk. The in-memory component is called frontline. It consists of a current data file, a spatial index (SAM) to index the non overlapped partitions,

and a set of TPR*-trees to index the records in the current data file. The current data file is situated in the temporal space without time strong log, and initialized when it is the first invocation of the spatio-temporal table's open method after system up. A hash index file association method used in the current data with UID as the hash key. It can provide fast recovery of records from the data file. Since there is a set of TPR*-tree, an additional pointer (slot id in SAM leaf page, or TPR*-tree ID, or pointer to the entry in TRP*-tree) can add to the tuple together with a latch for concurrency control. When a new page is allocate for the past data file, an entry is inserted in the sparse R*-tree with an vacant endtime in the lifetime.

2.2.2 Insertion Algorithm

The key to the performance of the insert algorithm in PCFI is the use of the hash index file, which maintains the last updated place of all moving objects. It is a cache of the last positions of all objects and is updated with the new place of the moving object. For simplify, the following method will use the 3-D sparse R*- tree to index the historical data file's page life, and the SAM leaf page's PID and entry id in the current data file tuple.

2.3 R PPF- tree

2.3.1 Structure

The data structures and algorithms connected with the RPPF-tree. we aim to detain and index the actual, real-world positions of the data objects across all of time. In temporal database terms, we believe the valid time of the objects' positions (although we do not allow general updates). We assume position samples for an object arrive in time order, and we expect the future transfer of the object by means of a linear function. When a new position sample arrives, we thus need to correct the guess that covers the period since when the last sample was received, and we need to re-predict the future position. This limited need for corrections enables us to support valid time by an comprehensive notion of partial persistence. Partial persistence transforms a linked data structure, termed an transient structure, into a corresponding data structure that retains and enables the querying of all past states (in the transaction-time sense) of the data being indexed.

2.3.2 Insertion Algorithm

The TPR-tree differs from the R*-tree in how its insertion algorithms group data points into nodes. The R*-tree aims to minimize the areas, overlaps, and margins of bounding rectangles when data objects are inserted into the index. The Update Algorithm. As input, the update algorithm takes the old entry eo, to be logically deleted (and corrected), and the new entry en, to be inserted. Both eo.t ⊣ and en. t ⊣ are infinite, en. t ⊣ is equal to the current time (CT), and eo. t ⊣ is equal to the time of the last update of this object. The update algorithm proceeds in the following five phases i.e in the deletion down phase, In the left phase, In the left-up phase, In the insertion down phase, and in the insertion up phase.

3. PERFORMANCE ANALYSES

3.1 Index sizes and space utilization

BBx – index

The BBx-index consists of nodes that in turn consist of entries, each of which is of the form hx rep; tstart; tend; pointer i. For leaf nodes, pointer points to the objects with the corresponding x rep, where x rep is obtained from the space-filling curve; tstart denotes the time when the object was inserted into the database. The roots of the trees are stored in an array, and they can be accessed efficiently according to their lifespan. This array is relatively small and can usually be stored in main memory.

PCFI+-Index

We implemented TPR*-tree with the node size equal to Xeon's cache size (16KB) as an additional indexing mechanism in Storm/NT. The current data file (hash index file) is implemented with the chunk size of 16KB and the pointers are TPR*-tree's IDs. In all our experiments, we use a buffer pool of 128MB. The disk page size used in all the experiments is 8KB. A value in the time dimension is represented using 14 bytes, and a value in the spatial dimension is represented using a 4 byte double.

R PPF- tree

workload of d = 0.2

While decreasing d leads to improved update performance, time slice query per-formance is naturally decreased. This is so because smaller values of d imply a smaller average fan out of the index as "seen" at the time of query. Nevertheless, the negative effect on query performance is less marked than the positive effect on update performance. Smaller values of d result in smaller index sizes.

3.2 Update Cost

BBx – index

Although migration introduces additional updates, the amortized cost is small in practice and is controllable. Query and update costs are measured in terms of node accesses.

Observe that the two indices have comparable performance as time passes, though the query cost of the BBx-index is slightly lesser than that of the other two. This is because the BBx-index needs to store less information than the others.

In order to investigate the performance degradation across time, we measure the update cost (amortized over insertion and deletion) of the BBx-index after every 20 timestamps over a 50K dataset. The BBx-index retains almost constant performance and is not affected by time. This is because given the key value, each deletion or insertion only needs to traverse one path from the top to the bottom of the tree.

PCFI+-Index

The SAM is used to index the non-overlapping partitions. Many spatial access methods can be used to implement the partition strategy. For example, the R-tree, R*-tree can be used when we partition the area into irregular shapes; the Grid-file, rough Grid file and quad tree can be used when we partition the area into regular rectangles. Partitioning the area into irregular cells raises another issue: how to send the partition information to the index manager? We can use a spatial table to store the partition data, and pass the table ID and field description when other modules invoke the index's open method. If the irregular partition is used, the index file can be to reduce the disk I/O cost of the partition information.

R PPF- tree

Double TPBRs support tightening, but they also have extra associated costs when compared with the two other kinds of indexing structures. Specifically, their update costs may be

higher. To learn how much the cost of updates can be reduced by increasing the size of the main memory buffer, we experimented with varying buffer sizes. As expected update costs decrease substantially when using a buffer of moderate size, compared to using no buffer or using a small buffer. However, very larger buffers are not very effective. This is because each update operation includes a number of write I/Os. Next, as expected, the search costs decrease for increasing buffer sizes, especially for future queries. This is because most of the live nodes become cached in the buffer.

3.3 Query Processing

The study shows that the BBx – index query performance is good than other two indexing structures. The figure 1 clearly shows that the time taken for the query retrieving of past data. X axis mention the number of objects and the y axis for execution time(ms). The graph clearly shows the BBx – index is best than other indexing structures.

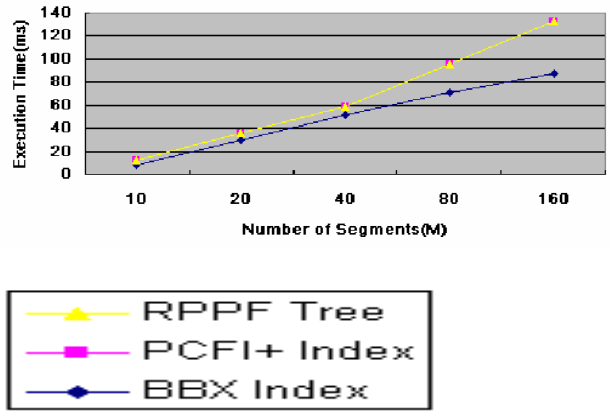

Figure 1 Retrieving Past data

The figure 2 shows that the time taken for the query retrieving of current data. X axis mention the number of objects and the y axis for execution time (ms). The graph clearly shows the

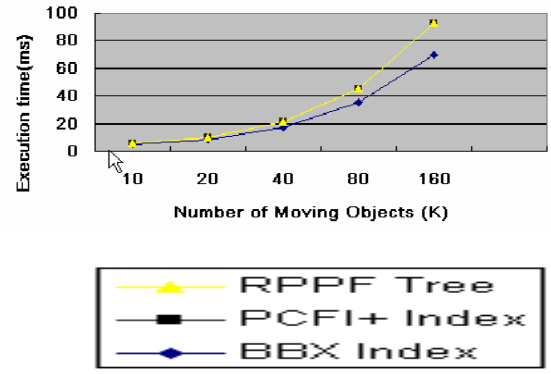

Figure 2 Retrieving current data

In BBx – index binary tree is used for node insertion and deletion. Because of binary tree method the time taken for node selection is too fast so the performance of BBx – index in better than other indexing structures.

4. CONCLUSION

A common objective was dealt in all the above three algorithms i.e indexing the data at all points of time(present, past and future). All the three algorithms are simple and well-designed data structure for indexing spatial data. This paper report a brief overview of BBx – index, PCFI+ index and RPPF – TREE indexing structure is provided. Moreover these three algorithms was released in same periods. In all the three cases the spatiotemporal data are based on abstracting the object's place as a function of time and also the methods taken to indexing in the spatiotemporal domain range are very miscellaneous. The study shows that the BBx – index is especially is very good for reducing storage space compared to an PCFI+ index and RPPF – TREE indexing structure and also less cost for storage and accesses the data. This evaluation can serve as a reference point for further evaluations and for the design of new, improved spatial temporal access methods.

5. REFERENCES

[1] Long-Van Nguyen-Dinh, Walid G. Aref, Mohamed F. Mokbel 2010. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[2] M. Pelanis, S. ˇ Saltenis, and C. Jensen. Indexing the past, present, and anticipated future positions of moving objects.TODS, 31(1):255–298, 2006.

[3] 3) Z.-H. Liu, X.-L. Liu, J.-W. Ge, and H.-Y. Bae. Indexing large moving objects from past to future with PCFI+-index. In COMAD, pages 131–137, 2005.

[4] V. Chakka, A. Everspaugh, and J. Patel. Indexing large trajectory data sets with SETI. In CIDR, 2003

[5] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In VLDB, 2003.

[6] D. Lin, C. Jensen, B. Ooi, and S. ˇ Saltenis. Efficient indexing of the historical, present, and future positions of moving objects. In MDM, pages 59–66, 2005.

[7] 7) C. Jensen, D. Lin, and B. Ooi. Query and update efficient B+-tree based indexing of moving objects. In VLDB, 2004.

[8] M. Mokbel, T. Ghanem, andW. G. Aref. Spatio-temporal access methods. IEEE Data Eng. Bull., 26(2):40–49, 2003.

[9] J. Ni and C. V. Ravishankar. PA-tree: A parametric indexing scheme for spatio-temporal trajectories. In SSTD, 2005.

[10] P. Zhou, D. Zhang, B. Salzberg, G. Cooperman, and G. Kollios. Close pair queries in moving object databases. In GIS, pages 2–11, 2005.

[11] S. Chen, B. Ooi, K. Tan, and M. Nascimento. ST2B-tree: A self-tunable spatio-temporal B+- tree index for moving objects. In SIGMOD, pages 29–42, 2008.

[12] P. K. Agarwal and C. M. Procopiuc. Advances in Indexing for Mobile Objects. IEEE Data Eng. Bull., 25(2): 25–34, 2002.

[13] G. Kollios, D. Gunopulos, V. J. Tsotras. On Indexing Mobile Objects. In Proc. PODS, pp. 261–272, 1999.