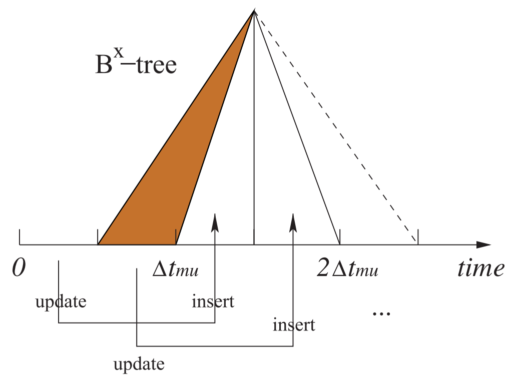
Indexing of Moving Objects, B+-Tree CE1

1 CHRISTIAN S. JENSEN[1], DAN LIN[2], BENG CHIN OOI[2]
2 [1] Department of Computer Science, Aalborg University,
3 Aalborg, Denmark
4 [2] Department of Computer Science, National University
5 of Singapore, Singapore, Singapore
6 csj@cs.aau.dk, lindan@comp.nus.edu.sg,
7 ooibc@comp.nus.edu.sg

8 Synonyms

9 B^x-tree



Query and Update Efficient B+-Tree-based Indexing of Moving Objects, Figure 1 An example of the B^x-tree

10 Definition

11 The B^x-tree [1] is a query and update efficient B^+-tree-
12 based index structure for moving objects which are rep-
13 resented as linear functions. The B^x-tree uses a lineariza-
14 tion technique to exploit the volatility of the data values
15 being indexed i. e., moving-object locations. Specifically,
16 data values are first partitioned according to their update
17 time and then linearized within the partitions according
18 to a space-filling curve, e. g., the Peano or Hilbert curve.
19 The resulting values are combined with their time partition
20 information and then indexed by a single B^+-tree. Figure 1
21 shows an example of the B^x-tree with the number of index
22 partitions equal to two within one maximum update inter-
23 val Δt_{mu}. In this example, there are maximum of three
24 partitions existing at the same time. After linearization,
25 object locations inserted at time 0 are indexed in partition
26 1, object locations updated during time 0 to 0.5 Δt_{mu} are
27 indexed in partition 2 and objects locations updated during
28 time 0.5 Δt_{mu} to time Δt_{mu} are indexed in partition 3 (as
29 indicated by arrows). As time elapses, repeatedly the first
30 range expires (shaded area), and a new range is appended
31 (dashed line). This use of rolling ranges enables the B^x-
32 tree to handle time effectively.

33 Historical Background

34 Traditional indexes for multidimensional databases, such
35 as the R-tree [2] and its variants were, implicitly or explic-
36 itly, designed with the main objective of supporting effi-
37 cient query processing as opposed to enabling efficient
38 updates. This works well in applications where queries
39 are relatively much more frequent than updates. Howev-
40 er, applications involving the indexing of moving objects
41 exhibit workloads characterized by heavy loads of updates
42 in addition to frequent queries.
43 Several new index structures have been proposed for
44 moving-object indexing. One may distinguish between
indexing of the past positions versus indexing of the cur- 45
rent and near-future positions of spatial objects. The B^x- 46
tree belongs to the latter category. 47
Past positions of moving objects are typically approximat- 48
ed by polylines composed of line segments. It is possible to 49
index line segments by R-trees, but the trajectory member- 50
ships of segments are not taken into account. In contrast to 51
this, the spatio-temporal R-tree [3] attempts to also group 52
segments according to their trajectory memberships, while 53
also taking spatial locations into account. The trajectory- 54
bundle tree [3] aims only for trajectory preservation, leav- 55
ing other spatial properties aside. Another example of this 56
category is the multi-version 3DR-tree [4], which com- 57
bines multi-version B-trees and R-trees. Using partial per- 58
sistence, multi-version B-trees guarantee time slice per- 59
formance that is independent of the length of the history 60
indexed. 61
The representations of the current and near-future posi- 62
tions of moving objects are quite different, as are the index- 63
ing challenges and solutions. Positions are represented as 64
points (constant functions) or functions of time, typical- 65
ly linear functions. The Lazy Update R-tree [5] aims to 66
reduce update cost by handling updates of objects that 67
do not move outside their leaf-level MBRs specially, and 68
a generalized approach to bottom-up update in R-trees has 69
recently been examined [6]. 70
Tayeb et al. [7] use PMR-quadtrees for indexing the future 71
linear trajectories of one-dimensional moving points as 72
line segments in (x, t)-space. The segments span the time 73
interval that starts at the current time and extends some 74
time into the future, after which time, a new tree must be 75
built. Kollis et al. [8] employ dual transformation tech- 76
niques which represent the position of an object moving 77
in a d-dimensional space as a point in a $2d$-dimensional 78
space. Their work is largely theoretical in nature. Based 79
on a similar technique, Patel et al. [9] have most recently 80

CE1 In order to help readers locate entries efficiently, the Editors-in-Chief changed a handful of entry titles, including your "Query and Update Efficient B+-Tree-based Indexing of Moving Objects". We're confident that this new title will be easy for readers to find and better suits the Table of Contents.
Please note that the pagination is not final; in the print version an entry will in general not start on a new page.
TS3 Please note that this figure will be printed in gray in the final version.

developed a practical indexing method, termed STRIPES, that supports efficient updates and queries at the cost of higher space requirements. Another representative indexes are the TPR-tree (time-parameterized R-tree) family of indexes (e. g., [10, 11]), which add the time parameters to bounding boxes in the traditional R-tree.

Scientific Fundamentals

Index Structure

The base structure of the B^x-tree is that of the B^+-tree. Thus, the internal nodes serve as a directory. Each internal node contains a pointer to its right sibling (the pointer is non-null if one exists). The leaf nodes contain the moving-object locations being indexed and corresponding index time.

To construct the B^x-tree, the key step is to map object locations to single-dimensional values. A space-filling curve is used for this purpose. Such a curve is a continuous path which visits every point in a discrete, multi-dimensional space exactly once and never crosses itself. These curves are effective in preserving proximity, meaning that points close in multidimensional space tend to be close in the one-dimensional space obtained by the curve. Current versions of the B^x-tree use the Peano curve (or Z-curve) and the Hilbert curve. Although other curves may be used, these two are expected to be particularly good according to analytical and empirical studies in [12]. In what follows, the value obtained from the space-filling curve is termed as the *x_value*.

An object location is given by $O = (\overrightarrow{x}, \overrightarrow{v})$, a position and a velocity, and an update time, or timestamp, t_u, where these values are valid. Note that the use of linear functions reduces the amount of updates to one third in comparison to constant functions. In a leaf-node entry, an object O updated at t_u is represented by a value $B^x value(O, t_u)$:

$$B^x value(O, t_u) = [index_partition]_2 \oplus [x_rep]_2 \qquad (1)$$

where *index_partition* is an index partition determined by the update time, *x_rep* is obtained using a space-filling curve, $[x]_2$ denotes the binary value of x, and $\oplus$ denotes concatenation.

If the timestamped object locations are indexed without differentiating them based on their timestamps, the proximity preserving property of the space-filling curve will be lost; and the index will also be ineffective in locating an object based on its *x_value*. To overcome such problems, the index is "partitioned" by placing entries in partitions based on their update time. More specifically, Δt_{mu} denotes the time duration that is the maximum duration in-between two updates of any object location. Then the time axis is partitioned into intervals of duration Δt_{mu}, and each such interval is sub-partitioned into n equal-length sub-intervals, termed *phases*. By mapping the update times in the same phase to the same so-called *label timestamp* and by using the label timestamps as prefixes of the representations of the object locations, index partitions are obtained, and the update times of updates determine the partitions they go to. In particular, an update with timestamp t_u is assigned a label timestamp $t_{lab} = \lceil t_u + \Delta t_{mu}/n \rceil_l$, where operation $\lceil x \rceil_l$ returns the nearest future label timestamp of x. For example, Fig. 1 shows a B^x-tree with $n = 2$. Objects with timestamp $t_u = 0$ obtain label timestamp $t_{lab} = 0.5\,\Delta t_{mu}$; objects with $0 < t_u \leq 0.5\Delta t_{mu}$ obtain label timestamp $t_{lab} = \Delta t_{mu}$; and so on. Next, for an object with label timestamp t_{lab}, its position at t_{lab} is computed according to its position and velocity at t_u. Then the space-filling curve is applied to this (future) position to obtain the second component of Eq. 1.

This mapping has two main advantages. First, it enables the tree to index object positions valid at different times, overcoming the limitation of the B^+-tree, which is only able to index a snapshot of all positions at the same time. Second, it reduces the update frequency compared to having to update the positions of all objects at each timestamp when only some of them need to be updated. The two components of the mapping function in Eq. 1 are consequently defined as follows:

$$index_partition = (t_{lab}/(\Delta t_{mu}/n) - 1)\, mod(n + 1)$$

$$x_rep = x_value(\overrightarrow{x} + \overrightarrow{v} \cdot (t_{lab} - t_u))$$

With the transformation, the B^x-tree will contain data belonging to $n+1$ phases, each given by a *label timestamp* and corresponding to a time interval. The value of n needs to be carefully chosen since it affects query performance and storage space. A large n results in smaller enlargements of query windows (covered in the following subsection), but also results in more partitions and therefore a looser relationship among object locations. In addition, a large n yields a higher space overhead due to more internal nodes.

To exemplify, let $n = 2$, $\Delta t_{mu} = 120$, and assume a Peano curve of order 3 (i. e., the space domain is 8×8). Object positions $O_1 = ((7, 2), (-0.1, 0.05))$, $O_2 = ((0, 6), (0.2, -0.3))$, and $O_3 = ((1, 2), (0.1, 0.1))$ are inserted at times 0, 10, and 100, respectively. The $B^x value$ for each object is calculated as follows.

Step 1: Calculate label timestamps and index partitions.

$$t^1_{lab} = \lceil(0+120/2)\rceil_l = 60,$$
$$index_partition^1 = 0 = (00)_2$$
$$t^2_{lab} = \lceil(10+120/2)\rceil_l = 120,$$
$$index_partition^2 = 1 = (01)_2$$
$$t^3_{lab} = \lceil(100+120/2)\rceil_l = 180,$$
$$index_partition^3 = 2 = (10)_2$$

Step 2: Calculate positions x_1, x_2, and x_3 at t^1_{lab}, t^2_{lab}, andt^3_{lab}, respectively.

$$x'_1 = (1, 5)$$
$$x'_2 = (2, 3)$$
$$x'_3 = (4, 1)$$

Step 3: Calculate Z-values.

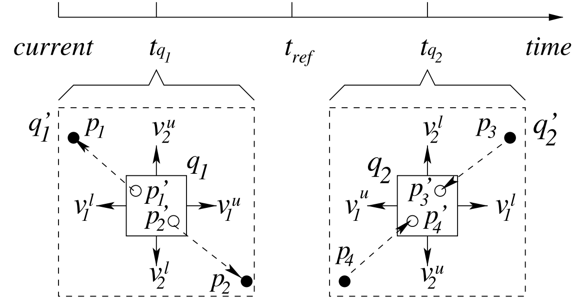
$$[Z_value(x'_1)]_2 = (010011)_2$$
$$[Z_value(x'_2)]_2 = (001101)_2$$
$$[Z_value(x'_3)]_2 = (100001)_2$$

Range Query Algorithm

A range query retrieves all objects whose location falls within the rectangular range $q = ([qx^l_1, qx^u_1], [qx^l_2, qx^u_2])$ at time t_q not prior to the current time ("l" denotes lower bound, and "u" denotes upper bound).

A key challenge is to support predictive queries, i. e., queries that concern future times. Traditionally, indexes that use linear functions handle predictive queries by means of bounding box enlargement (e. g., the TPR-tree). Whereas, the B^x-tree uses query-window enlargement. Since the B^x-tree stores an object's location as of some time after its update time, the enlargement involves two cases: a location must either be brought back to an earlier time or forward to a later time. Consider the example in Fig. 2, where t_{ref} denotes the time when the locations of four moving objects are updated to their current value index, and where predictive queries q_1 and q_2 (solid rectangles) have time parameters t_{q1} and t_{q2}, respectively.

The figure shows the stored positions as solid dots and positions of the two first objects at t_{q1} and the positions of the two last at t_{q2} as circles. The two positions for each object are connected by an arrow. The relationship between the two positions for each object is $p'_i = p_i + \overrightarrow{v} \cdot (t_q - t_{ref})$. The first two of the four objects, thus, are in the result of the first query, and the last two objects are in the result of the second query. To obtain this result, query rectangle q_1 needs to be enlarged to q'_1 (dashed). This is achieved by attaching maximum speeds to the sides of



Query and Update Efficient B+-Tree-based Indexing of Moving Objects, Figure 2 Query window enlargement

q_1 : v^l_1, v^l_2, v^u_1, andv^u_2. For example, v^u_1 is obtained as the largest projection onto the x-axis of a velocity of an object in q'_1. For q_2, the enlargement speeds are computed similarly. For example, v^u_2 is obtained by projecting all velocities of objects in q'_2 onto the y-axis; v^u_2 is then set to the largest speed multiplied by –1.

The enlargement of query $q = ([qx^l_1, qx^u_1], [qx^l_2, qx^u_2])$ is given by query $q' = ([eqx^l_1, eqx^u_1], [eqx^l_2, eqx^u_2])$:

$$eqx^l_i = \begin{cases} qx^l_i + v^l_i \cdot (t_{ref} - t_q) & \text{if } t_q < t_{ref} \\ qx^l_i + v^u_i \cdot (t_q - t_{ref}) & \text{otherwise} \end{cases} \quad (2)$$

$$eqx^u_i = \begin{cases} qx^u_i + v^u_i \cdot (t_{ref} - t_q) & \text{if } t_q < t_{ref} \\ qx^u_i + v^l_i \cdot (t_q - t_{ref}) & \text{otherwise} \end{cases} \quad (3)$$

The implementation of the computation of enlargement speeds proceeds in two steps. They are first set to the maximum speeds of all objects, thus a preliminary q' is obtained. Then, with the aid of a two-dimensional histogram (e. g., a grid) that captures the maximum and minimum projections of velocities onto the axes of objects in each cell, the final enlargement speed in the area where the query window resides is obtained. Such a histogram can easily be maintained in main memory.

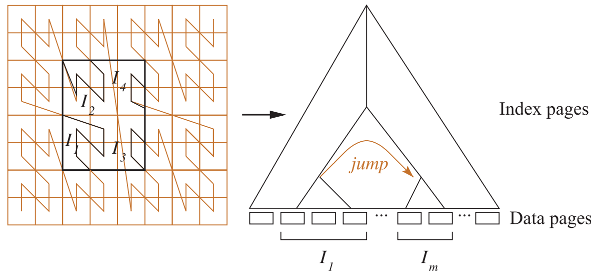
Next, the partitions of the B^x-tree are traversed to find objects falling in the enlarged query window q'. In each partition, the use of a space-filling curve means that a range query in the native, two-dimensional space becomes a set of range queries in the transformed, one-dimensional space (see Fig. 3); hence multiple traversals of the index result. These traversals are optimized by calculating the start and end points of the one-dimensional ranges and traverse the intervals by "jumping" in the index.

k Nearest Neighbor Query Algorithm

Assuming a set of $N>k$ objects and given a query object with position $q = (qx_1, qx_2)$, the k nearest neighbor query (kNN query) retrieves k objects for which no other objects

TS4 Please note that this figure will be printed in gray in the final version.

Editor's or typesetter's annotations (will be removed before the final TEX run)



Query and Update Efficient B$^+$-Tree-based Indexing of Moving Objects, Figure 3 Jump in the index

are nearer to the query object at time t_q not prior to the current time.

This query is computed by iteratively performing range queries with an incrementally enlarged search region until k answers are obtained. First, a range R_{q1} centered at q with extension $r_q = D_k/k$ is constructed. D_k is the estimated distance between the query object and its k'th nearest neighbor; D_k can be estimated by the following equation [13]:

$$D_k = \frac{2}{\sqrt{\pi}} \left[1 - \sqrt{1 - \left(\frac{k}{N}\right)^{1/2}} \right] .$$

The range query with range R_{q1} at time t_q is computed, by enlarging it to a range R'_{q1} and proceeding as described in the previous section. If at least k objects are currently covered by R'_{q1} and are enclosed in the inscribed circle of R_{q1} at time t_q, the kNN algorithm returns the k nearest objects and then stops. It is safe to stop because all the objects that can possibly be in the result have been considered. Otherwise, R_{q1} is extended by r_q to obtain R_{q2} and an enlarged window R'_{q2}. This time, the region $R'_{q2} - R'_{q1}$ is searched and the neighbor list is adjusted accordingly. This process is repeated until we obtain an R_{qi} so that there are k objects within its inscribed circle.

Continuous Query Algorithm

The queries considered so far in this section may be considered as one-time queries: they run once and complete when a result has been returned. Intuitively, a continuous query is a one-time query that is run at each point in time during a time interval. Further, a continuous query takes a *now*-relative time $now + \Delta t_q$ as a parameter instead of the fixed time t_q. The query then maintains the result of the corresponding one-time query at time $now + \Delta t_q$ from when the query is issued at time t_{issue} and until it is deactivated.

Such a query can be supported by a query q_e with time interval $[t_{issue} + \Delta t_q, t_{issue} + \Delta t_q + l]$ ("l" is a time interval) [14]. Query q_e can be computed by the algorithms presented previously, with relatively minor modifications: (i) use the end time of the time interval to perform forward enlargements, and use the start time of the time interval for backward enlargements; (ii) store the answer sets during the time interval. Then, from time t_{issue} to $t_{\text{issue}} + l$, the answer to q_l is maintained during update operations. At $t_{\text{issue}} + l$, a new query with time interval $[t_{issue} + \Delta t_q + l, t_{issue} + \Delta t_q + 2l]$ is computed.

A continuous range query during updates can be maintained by adding or removing the object from the answer set if the inserted or deleted object resides in the query window. Such operations only introduce CPU cost.

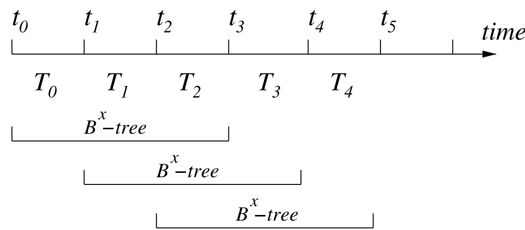
The maintenance of continuous kNN queries is somewhat more complex. Insertions also only introduce CPU cost: an inserted object is compared with the current answer set. Deletions of objects not in the answer set does not affect the query. However, if a deleted object is in the current answer set, the answer set is no longer valid. In this case, a new query with a time interval of length l at the time of the deletion is issued. If the deletion time is t_{del}, a query with time interval $[t_{del} + \Delta t_q, t_{del} + \Delta t_q + l]$ is triggered at t_{del}, and the answer set is maintained from t_{del} to $t_{del} + l$.

The choice of the "optimal" l value involves a trade-off between the cost of the computation of the query with the time interval and the cost of maintaining its result. On the one hand, a small l needs to be avoided as this entails frequent recomputations of queries, which involve a substantial I/O cost. On the other hand, a large l introduces a substantial cost: Although computing one or a few queries is cost effective in itself, the cost of maintaining the larger answer set must also be taken into account, which may generate additional I/Os on each update. Note that maintenance of continuous range queries incur only CPU cost. Thus, a range query with a relatively large l is computed such that l is bounded by $\Delta t_{mu} - \Delta t_q$ since the answer set obtained at t_{issue} is no longer valid at $t_{\text{issue}} + \Delta t_{mu}$. For the continuous kNN queries, l needs to be carefully chosen.

Update, Insertion, and Deletion

Given a new object, its index key is calculated according to Eq. 1, and then insert it into the B^x-tree as in the B$^+$-tree. To delete an object, an assumption is made that the positional information for the object used at its last insertion and the last insertion time are known. Then its index key is calculated and the same deletion algorithm as in the B$^+$-tree is employed. Therefore, the B^x-tree directly inherits the good properties of the B$^+$-tree, and efficient update performance is expected.

However, one should note that update in the B^x-tree does differ with respect to update in the B$^+$-tree. The B^x-tree



Query and Update Efficient B+-Tree-based Indexing of Moving Objects, Figure 4 B^x-tree evolution

only updates objects when their moving functions have been changed. This is realized by clustering updates during a certain period to one time point and maintaining several corresponding sub-trees. The total size of the three subtrees is equal to that of one tree indexing all the objects.

In some applications, there may be some object positions that are updated relatively rarely. For example, most objects may be updated at least each 10 minutes, but a few objects are updated once a day. Instead of letting outliers force a large maximum update interval, a "maximum update interval" within which a high percentage of objects have been updated is used. Object positions that are not updated within this interval are "flushed" to a new partition using their positions at the label timestamp of the new partition. In the example shown in Fig. 4, suppose that some object positions in T_0 are not updated at the time when T_0 expires. At this time, these objects are moved to T_2. Although this introduces additional update cost, the (controllable) amortized cost is expected to be very small since outliers are rare. The forced movement of an object's position to a new partition does not cause any problem with respect to locating the object, since the new partition can be calculated based on the original update time. Likewise, the query efficiency is not affected.

Key Applications

With the advances in positioning technologies, such as GPS, and rapid developments of wireless communication devices, it is now possible to track continuously moving objects, such as vehicles, users of wireless devices and goods. The B^x-tree can be used in a number of emerging applications involving the monitoring and querying of large quantities of continuous variables, e. g., the positions of moving objects. In the following, some of these applications are discussed.

Location-Based Service

A traveller comes to a city that he is not familiar with. To start with, he sends his location by using his PDA or smart phone (equipped with GPS) to a local server that provides location-based services. Then the service provider can answer queries like "where is the nearest restaurant (or hotel)?" and can also help to dispatch a nearby taxi to the traveller.

A driver can also benefit from the location-based services. For example, he can ask for a nearest gas station or motel when he is driving.

Traffic Control

If the moving objects database stores information about locations of vehicles, it may be able to predict the possible congestion in near future. To avoid the congestion, the system can divert some vehicles to alternate routes in advance.

For air traffic control, the moving objects database system can retrieve all the aircrafts within a certain region and prevent a possible collision.

E-commerce

In these applications, stores send out advertisements or e-coupons to vehicles passing by or within the store region.

Digital Game

Another interesting example is location-based digital game where the positions of the mobile users play a central role. In such games, players need to locate their nearest neighbors to fulfill "tasks" such as "shooting" other close players via their mobile devices.

Battle Field

The moving object database technique is also very important in the military. With the help of the moving object database techniques, helicopters and tanks in the battlefield may be better positioned and mobilized to the maximum advantage.

RFID Application

Recently, applications using radio frequency identification (RFID) has received much interest. RFID enables data to be captured remotely via radio waves and stored on electronic tags embedded in their carriers. A reader (scanner) is then used to retrieve the information. In a hospital application, RFIDs are tagged to all patients, nurses and doctors, so that the system can keep a real-time tracking of their movements. If there is an emergency, nurses and doctors can be sent to the patients more quickly.

Future Directions

Several promising directions for future work exist. One could be the improvement of the range query performance in the B^x-tree since the current range query algorithm uses the strategy of enlarging query windows which may incur some redundant search. Also, the use of the B^x-tree for the processing of new kinds of queries can be considered. Another direction is the use of the B^x-tree for other continuous variables than the positions of mobile service users. Yet another direction is to apply the linearization technique to other index structures.

Cross References

► Indexing, BDual-tree
► Indexing the Positions of Continuously Moving Objects

Recommended Reading CE5

1. Jensen, C.S., Lin, D., Ooi, B.C.: Query and update efficient B+-tree based indexing of moving objects. In: Proc. VLDB, pp. 768–779 (2004)
2. Guttman, A.: R-trees: A Dynamic Index Structure for Spatial Searching. In: Proc. ACM SIGMOD, pp. 47–57 (1984)
3. Pfoser, D., Jensen, C.S., Theodoridis, Y.: Novel approaches in query processing for moving objects. In: Proc. VLDB, pp. 395–406 (2000)
4. Tao, Y., Papadias, D.: MV3R-tree: a spatio-temporal access method for timestamp and interval queries. In: Proc. VLDB, pp. 431–440 (2001)
5. Kwon, D., Lee, S., Lee, S.: Indexing the current positions of moving objects using the lazy update R-tree. In: Proc. MDM, pp. 113–120 (2002)
6. Lee, M.L., Hsu, W., Jensen, C.S., Cui, B., Teo, K.L.: Supporting frequent updates in R-trees: a bottom-up approach. In: Proc. VLDB, pp. 608–619 (2003)
7. Tayeb, J., Ulusoy, O., Wolfson, O.: A quadtree based dynamic attribute indexing method. Computer J. **41**(3):185–200 (1998)
8. Kollios, G., Gunopulos, D., Tsotras, V.J.: On indexing mobile objects. In: Proc. PODS, pp. 261–272 (1999)
9. Patel, J.M., Chen Y., Chakka V.P.: STRIPES: An efficient index for predicted trajectories. In: Proc. ACM SIGMOD, (2004) (to appear)
10. Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the positions of continuously moving objects. In: Proc. ACM SIGMOD, pp. 331–342 (2000)
11. Tao, Y., Papadias, D., Sun, J.: The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In: Proc. VLDB, pp. 790–801 (2003)
12. Moon, B., Jagadish, H.V., Faloutsos, C., Saltz J.H.: Analysis of the clustering properties of the hilbert space-filling curve. IEEE TKDE **13**(1):124–141 (2001)
13. Tao, Y., Zhang, J., Papadias, D., Mamoulis, N.: An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. IEEE TKDE **16**(10):1169–1184 (2004)
14. Benetis, R., Jensen, C.S., Karciauskas, G., Saltenis, S.: Nearest neighbor and reverse nearest neighbor queries for moving objects. In: Proc. IDEAS, pp. 44–53 (2002)

Maximum Update Interval in Moving Objects Databases

Christian S. Jensen[1], Dan Lin[2], Beng Chin Ooi[2]
[1] Department of Computer Science, Aalborg University, Aalborg, Denmark
[2] Department of Computer Science, National University of Singapore, Singapore, Singapore
csj@cs.aau.dk, lindan@comp.nus.edu.sg, ooibc@comp.nus.edu.sg

Synonyms

Maximum update interval

Definition

The maximum update interval in moving objects databases denotes the maximum time duration in-between two subsequent updates of the position of any moving object. In some applications, a variation of the maximum update interval denotes the time duration within which a high percentage of objects have been updated.

Main Text

In moving objects databases, there exist a population of moving objects, where each object is usually assumed to be capable of transmitting its current location to a central server. A moving object transmits a new location to the server when the deviation between its real location and its server-side location exceeds a threshold, dictated by the services to be supported. In general, the deviation between the real location and the location predicted by the server tends to increase as time passes. Even the deviation does not increase, it is also necessary to inform the server periodically that the object still exists in the system. In keeping with this, a *maximum update interval* is defined as a problem parameter that denotes the maximum time duration in-between two updates of the position of any moving object. This definition is very helpful to index and application development especially those with functions of future prediction. Given such a time interval, trajectories of objects beyond their maximum update interval when the objects will definitely be updated usually do not need to be considered. In other words, the maximum update interval gives an idea of a time period of validity of current object information.

Cross References

CE2

CE5 We changed the order of the references. Please confirm. Thank you.
Please note that the pagination is not final; in the print version an entry will in general not start on a new page.
CE2 Please provide a section with cross references. See "Contents" on http://refworks.springer.com/geograph/ for related topics.

Editor's or typesetter's annotations (will be removed before the final TEX run)