Bx-tree

From Wikipedia, the free encyclopedia
(Redirected from Bx-tree Moving Object Index)

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Српски / srpski
Edit links

In computer science, the **B^x tree** is a query and update efficient B+ tree-based index structure for moving objects.

Contents

Index structure [edit]

The base structure of the B^x-tree is a B+ tree in which the internal nodes serve as a directory, each containing a pointer to its right sibling. In the earlier version of the B^x-tree,[1] the leaf nodes contained the moving-object locations being indexed and corresponding index time. In the optimized version,[2] each leaf node entry contains the id, velocity, single-dimensional mapping value and the latest update time of the object. The fanout is increased by not storing the locations of moving objects, as these can be derived from the mapping values.

Utilizing the B+ tree for moving objects [edit]

As for many other moving objects indexes, a two-dimensional moving object is modeled as a linear function as O = ((x, y), (vx, vy), t), where (x, y) and (vx, vy) are location and velocity of the object at a given time instance *t*, i.e., the time of last update. The B+ tree is a structure for indexing single-dimensional data. In order to adopt the B+ tree as a moving object index, the B^x-tree uses a linearization technique which helps to integrate objects' location at time *t* into single dimensional value. Specifically, objects are first partitioned according to their update time. For objects within the same partition, the B^x-tree stores their locations at a given time which are estimated by linear interpolation. By doing so, the B^x-tree keeps a consistent view of all objects within the same partition without storing the update time of an objects.

Secondly, the space is partitioned by a grid and the location of an object is linearized within the partitions according to a space-filling curve, e.g., the Peano or Hilbert curves.

An example of the B^x-tree with the number of index partitions equal to 2 within one maximum update interval tmu. In this example, there are maximum three partitions existing at the same time. After linearization, object locations inserted at time 0 are indexed in partition 0 with label timestamp 0.5 tmu, object locations updated during time 0 to 0.5 tmu are indexed in partition 1 with label timestamp tmu, and so on (as indicated by arrows). As time elapses, repeatedly the first range expires (shaded area), and a new range is appended (dashed line).

Finally, with the combination of the partition number (time information) and the linear order (location information), an object is indexed in B^x-tree with a one-dimensional index key B^xvalue:

Here index-partition is an index partition determined by the update time and xrep is the space-filling curve value of the object position at the indexed time, [ ] denotes the binary value of x, and "+" means concatenation.

Given an object O ((7, 2), (-0.1,0.05), 10), tmu = 120, the B^xvalue for O can be computed as follows.

1. O is indexed in partition 0 as mentioned. Therefore, indexpartition = $(00)_2$.
2. O's position at the label timestamp of partition 0 is (1,5).
3. Using Z-curve with order = 3, the Z-value of O, i.e., xrep is $(010011)_2$.
4. Concatenating indexpartition and xrep, B^xvalue $(00010011)_2$=19.
5. Example O ((0,6), (0.2, -0.3),10) and tmu=120 then O's position at the label timestamp of partition: ???

Insertion, update and deletion [edit]

Given a new object, its index key is computed and then the object is inserted into the B^x-tree as in the B+ tree. An update consists of a deletion followed by an insertion. An auxiliary structure is employed to keep the latest key of each index so that an object can be deleted by searching for the key. The indexing key is computed before affecting the tree. In this way, the B^x-tree directly inherits the good properties of the B+ tree, and achieves efficient update performance.

Queries [edit]

Range query [edit]

A range query retrieves all objects whose location falls within the rectangular range [ ] at time [ ] not prior to the current time.

The B^x-tree uses query-window enlargement technique to answer queries. Since the B^x-tree stores an object's location as of sometime 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. The main idea is to enlarge the query window so that it encloses all objects whose positions are not within query window at its label timestamp but will enter the query window at the query timestamp.

After the enlargement, the partitions of the B^x-tree need to be traversed to find objects falling in the enlarged query window. 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.[1]

To avoid excessively large query region after expansion in skewed datasets, an optimization of the query algorithm exists,[3] which improves the query efficiency by avoiding unnecessary query enlargement.

K nearest neighbor query [edit]

K nearest neighbor query is computed by iteratively performing range queries with an incrementally enlarged search region until k answers are obtained. Another possibility is to employ similar querying ideas in The iDistance Technique.

Other queries [edit]

The range query and K Nearest Neighbor query algorithms can be easily extended to support interval queries, continuous queries, etc.[2]

Adapting relational database engines to accommodate moving objects [edit]

Since the B^x-tree is an index built on top of a B+ tree index, all operations in the B^x-tree, including the insertion, deletion and search, are the same as those in the B+ tree. There is no need to change the implementations of these operations. The only difference is to implement the procedure of deriving the indexing key as a stored procedure in an existing DBMS. Therefore the B^x-tree can be easily integrated into existing DBMS without touching the kernel.

SpADE[4] is moving object management system built on top of a popular relational database system MySQL, which uses the B^x-tree for indexing the objects. In the implementation, moving object data is transformed and stored directly on MySQL, and queries are transformed into standard SQL statements which are efficiently processed in the relational engine. Most importantly, all these are achieved neatly and independently without infiltrating into the MySQL core.

Performance tuning [edit]

Potential problem with data skew [edit]

The B^x tree uses a grid for space partitioning while mapping two-dimensional location into one-dimensional key. This may introduce performance degradation to both query and update operations while dealing with skewed data. If grid cell is oversize, many objects are contained in a cell. Since objects in a cell are indistinguishable to the index, there will be some overflow nodes in the underlying B+ tree. The existing of overflow pages not only destroys the balancing of the tree but also increases the update cost. As for the queries, for the given query region, large cell incurs more false positives and increases the processing time. On the other hand, if the space is partitioned with finer grid, i.e. smaller cells, each cell contains few objects. There is hardly overflow pages so that the update cost is minimized. Fewer false positives are retrieved in a query. However, more cells are needed to be searched. The increase in the number of cells searched also increases the workload of a query.

Index tuning [edit]

The ST^2B-tree [5] introduces a self-tuning framework for tuning the performance of the B^x-tree while dealing with data skew in space and data change with time. In order to deal with data skew in space, the ST^2B-tree splits the entire space into regions of different object density using a set of reference points. Each region uses an individual grid whose cell size is determined by the object density inside of it.

The B^x-tree have multiple partitions regarding different time intervals. As time elapsed, each partition grows and shrinks alternately. The ST^2B-tree utilizes this feature to tune the index online in order to adjust the space partitioning to make itself accommodate to the data changes with time. In particular, as a partition shrinks to empty and starts growing, it chooses a new set of reference points and new grid for each reference point according to the latest data density. The tuning is based on the latest statistics collected during a given period of time, so that the way of space partitioning is supposed to fit the latest data distribution best. By this means, the ST^2B-tree is expected to minimize the effect caused by data skew in space and data changes with time.

See also [edit]

- B+ tree
- Hilbert curve
- Z-order (curve)

References [edit]

1. ^ a b Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and Update Efficient B+tree based Indexing of Moving Objects . In Proceedings of 30th International Conference on Very Large Data Bases (VLDB), pages 768-779, 2004.
2. ^ a b Dan Lin. Indexing and Querying Moving Objects Databases , PhD thesis, National University of Singapore, 2006.
3. ^ Jensen, C.S., D. Tiesyte, N. Tradisauskas, Robust B+-Tree-Based Indexing of Moving Objects, in Proceedings of the Seventh International Conference on Mobile Data Management , Nara, Japan, 9 pages, May 9–12, 2006.
4. ^ SpADE: A SPatio-temporal Autonomic Database Engine for location-aware services.
5. ^ Su Chen, Beng Chin Ooi, Kan-Lee. Tan, and Mario A. Nacismento, ST2B-tree: A Self-Tunable Spatio-Temporal B+-tree for Moving Objects . In Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD), page 29-42, 2008.

v · t · e **Tree data structures**

Search trees (dynamic sets/associative arrays)	2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · **B^x** · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T · Treap · UB · Weight-balanced
Heaps	Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · Van Emde Boas
Tries	Hash · Radix · Suffix · Ternary search · X-fast · Y-fast
Spatial data partitioning trees	BK · BSP · Cartesian · Hilbert R · *k*-d (implicit *k*-d) · M · Metric · MVP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X
Other trees	Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary · Left-child right-sibling · Link/cut · Log-structured merge · Merkle · PQ · Range · SPQR · Top

Categories: B-tree

This page was last modified on 10 February 2015, at 00:27.