Spatio-temporal access methods

Tong Jia
Xiangyu Li
Yongyi Liu
Contents

• Indexing the Past
• Indexing the Current
• Indexing the Future
Indexing the past

Multi-dimensional structures

Tong Jia
PH-Tree
(PATRICIA-hypercube-tree)

- A multi-dimensional data structure
- Extends both the Quad-tree and the PATRICIA-trie
- Optimize the search performance and the space utilization
- Indexing large amounts of multi-dimensional data.
Quad-tree

- Efficiently store data of points on a two-dimensional space.
- Each node has at most four children.
- Rarely used outside 2D or 3D problems
PATRICIA-Trie (prefix tree)

- Trie: strings are stored in a prefix-sharing method—Much more space efficient than storing each key individually.
- PATRICIA trees are radix trees with radix equals 2
- In general, any kind of data can be stored in such a tree by taking the bit representation of the data
PH-Tree

- k-dimensional object
- Partitions the space across all dimensions at any given node.
- Serializes the attributes of the indexed objects using binary representation.
- Can be seen as a hyper-cube of size $2^k$
- Is essentially a quadtree that uses hyper-cubes, prefix-sharing and bit-stream storage.
Advantages

• Makes access virtually independent of the order.
• Reduce the number of nodes in the tree
• The maximum depth is independent of $k$ and equal to the number of bits in the longest stored value.
Advantages

• No need for rebalancing because it’s unbalanced.
• Stable with respect to insert or delete operations.
• This is useful for concurrency when stored on disk--limits the number of pages that need to be rewritten.
1D PH-Tree

- resembles the binary PATRICIA trie.
- The value is stored in its binary representation as a bit-string.
- The first bit is stored in the root node. (In the 1D-case, all entries starting with a 0 can be found below the left box, all starting with a 1 can be found below the right box.)
- The depth of the trees is thus limited to 4.

Figure 1: A sample 1D PH-tree with one 4-bit entry (a) and two 4-bit entries (b)
1D PH-Tree

Figure 1: A sample 1D PH-tree with one 4-bit entry (a) and two 4-bit entries (b)

- Entries that are attached to an array field without further sub-nodes, such as the 010, are called a **postfix**.
- A second value 0001 has been **added** to the tree in Figure 1b.
2D PH-tree

Figure 2: A sample 2D PH-tree with three 4-bit entries: (0001, 1000), (0011, 1000), (0011, 1010)
Indexing the current

Xiangyu Li
Challenges

• Frequent Updating
Challenges

• Frequent Updating
  – Locating -> Top-down
  – Deletion -> Merging
  – Insertion -> Splitting

Fig. 1 An Example of Location Update
Challenges

• **Frequent Updating**
  – Locating -> Top-down
  – Deletion -> Merging
  – Insertion -> Splitting

• **Inefficient** -> **Real-time Response**
Challenges

• Frequent Updating
  – Locating -> Top-down
  – Deletion -> Merging
  – Insertion -> Splitting

• Inefficient -> Real-time Response X

• Caching -> Reduce I/O cost -> Real-time Response
Challenges

• Frequent Updating
  – Locating -> Top-down
  – Deletion -> Merging
  – Insertion -> Splitting

• Inefficient -> Real-time Response X

• Caching -> Reduce I/O cost -> Real-time Response

• 1. RUM+-tree (R-tree-based)

• 2. DIME (Disposable Index for Moving Objects)
RUM+-tree

Fig. 2. Structure of RUM+-tree

- Hash Table(with Obj ID) + Update Memo
RUM+-tree

• Hash Table (with Obj ID)
• -> directly locate objects

• Update Memo
• -> cache the costly modification
RUM+-tree

• With Update Memo
• Update:
  – Cheap one -> do
  – Costly -> cache

Lazy Update + Batch -> Avoid Frequent split/merge
RUM+-tree

(a)

Create new version

(b)

Split

R0

(c)
Fig. 2. Structure of RUM+-tree
DIME: Disposable Index for Moving Objects

• Do not modify the index at all!
DIME: Disposable Index for Moving Objects

- Do not modify the index at all!
- Modify the index ->
- Detach a whole chunk of the index

Fig. 2 Location Update on Disposable Index
**DIME**

**Dispersed Index**

**Fig. 2 Location Update on Disposable Index**

<table>
<thead>
<tr>
<th>Concept</th>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum time interval</td>
<td>$\Delta t_{mn}$</td>
<td>Maximum time interval for moving objects to update locations</td>
</tr>
<tr>
<td>Phase</td>
<td>$\Delta t = \Delta t_{mn}/n$</td>
<td>Time interval to construct an indexing component</td>
</tr>
<tr>
<td>Component</td>
<td>$C_t$</td>
<td>Indexing component constructed by timestamp $t$</td>
</tr>
<tr>
<td>Lifetime</td>
<td>$L_t = (n+1)\Delta t$</td>
<td>Time period from constructing an indexing component to disposing it</td>
</tr>
</tbody>
</table>
DIME

$t_0$

P1, P2, P3, P4, P5, P6, P7, P8, P9
n equals to 2.

At t2, the components of t0 need to be disposed.
Indexing the Future
Yongyi Liu
Indexing the Future Based on Underlying Road-Network

“Predictive Tree: An Efficient Index for Predictive Queries on Road Networks”
Challenges

- functional limitations
  1. distance measure
  2. training data
  3. flexibility

- performance deficiencies
The implementation system
-iRoad System Architecture

State Manager: R-tree, trajectory buffer, predictive tree
Predictive Tree builder: the moving object's trajectory buffer, the moving object's current predictive tree, the tunable parameters
Query processor
Fig. 2. Example Of The Proposed Index Structure
Predictive Tree Construction

• Initialization

visited nodes list: record nodes processed so far

min-heap: order the nodes based on distance to the root

• Expansion

continuously pop the root from the min-heap and expand the predictive tree
Initialization

Expansion

Fig. 4. Example of Constructing And Expanding The Predictive Tree Started At Node A.
Predictive Tree Maintenance

Main Idea:
update the root and prune the unnecessary part

Fig. 5. Demonstration Example For An Object Trip And Predictive Tree Maintenance
basic query and extensions

• predictive point query
  - to find out the moving objects with their corresponding probabilities that are expected to be around a specified query node in the road network within a future time period

extension to range queries, aggregate queries, KNN