Abstract

This paper has two parts. The first part presents the join indexes. It covers the most two join indexing, which are foreign column join index and multitable join index. The second part introduces some of the indexing techniques used for High-Dimensional data. The two indexing methods discussed are the X-tree, the TV-tree. Each section within the second part covers one of the mentioned indexing techniques. It introduces the indexing method, and illustrates its tree structure along with its different algorithms.

I. Join indexes

The join operations are highly used to execute SQL queries in almost all relational DBMS. DBMSs implemented multitude algorithms to perform joins efficiently. Nested loop join, merge join, hybrid join, and hash join are some of these algorithms. But, a new approach called join indexing start to be adopted by new or updated DBMSs because it has offered significant additional acceleration for many queries.

A join index provides a means to translate restrictions on columns of one table to restrictions on another table, through a commonly encountered join

1. The foreign column join index

I will explain this indexing technique through an example. Let us suppose two tables CUSTOMER and SALES. These tables are joined by the cid primary key of the CUSTOMER table with the foreign key cid of the SALES table.

So, we want to perform the following query:

```
select sum(s.dollar_sales) from CUSTOMER c, SALES s
where c.gender = 'M' and c.state = 'IL'
and c.hobby in ('jogging', 'racquetball', 'squash', . . .)
and c.cid = s.cid and s.department = 'sports'
group by c.familyincome;
```
Logically there is only one customer for each sale. Therefore; we can select a set of sales rows that have male customers. This can be presented as a bitmap. Similarly, we can restrict the SALES rows to those that have female customers and create another bitmap. Up to now, we create an index in SALES with two values, ‘M’ and ‘F’, and provide these two bitmaps for these values. We created a gender index on SALES, even though there is no gender column in the SALES table. We have used a column of CUSTOMER to create this index, depending on a join between the two tables to do this. This index is a foreign column join (FCJ) index. The FCJ index is a variant form of the bitmap join index introduced. We can create FCJ indexes for state and hobby on the SALES table. With these join indexes all in place, we note that we can now perform the restriction, grouping, and aggregation of the select statement entirely in terms of rows and indexes of the SALES table alone, without any join to the CUSTOMER table being performed at all.

2. The multitable join index

A multitable join (MTJ) index allows us to translate restrictions on columns of several tables at once to a restriction on another table joined to them. This technique is not flexible for that reason MTJ indexes are only used in extremely specific situations. Here again, I will an example to explain the MTJ index technique. We have the following star schema:

Now assume that we needs to be performed frequently the following SQL statement:

```sql
select sum(dollar_sales), sum(unit_sales)
from SALES s, CUSTOMERS c, PRODUCTS p, TIME t
where s.cid = c.cid and s.pid = p.pid
and s.day = t.day
and t.month = 'May95' and p.package_type = 'box'
and c.gender = 'M';
```
An MTJ index is constructed by taking each row of the SALES table and determining the value triple of t.month, p.package_type, and c.gender for that row. Then we will have an entry in our MTJ index with this triple of values concatenated (joined together) in sequence as the index value, and this entry will contain a Row Identifier (RID) list or bitmap representation of the set of rows in SALES that had this triple of values.

The efficiency gain of an MTJ index over three FCJ indexes is not tremendously significant and also the MTJ is inflexible, because if we want to use the MTJ we must known the common queries in our workload which is not the case in all the time.

II Indexing High Dimensional data

1. Introduction

Classical and traditional database systems deal with one-dimensional data set (number and string). Databases experts developed a quiet number of advanced techniques that proved its efficiency in data storage, update, and retrieval. The most commonly used indexing technique is B-tree. But in the past decade, more and more advanced applications emerged to the market and require manipulation of multidimensional data. For example, Geographical Information System (GIS), Multimedia databases, Medical databases, Time series and so on.

<table>
<thead>
<tr>
<th>Dimensionality</th>
<th>Operations</th>
<th>Indexing Techniques</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert, delete, exact/partial match</td>
<td>B-tree, hash indexing</td>
<td>Data warehouse</td>
</tr>
<tr>
<td>2-10</td>
<td>insert, delete, exact match, spatial-temporal search</td>
<td>quad-tree, grid-file, K-D B-tree, R-tree, R+-tree, R*-tree</td>
<td>GIS</td>
</tr>
<tr>
<td>&gt;10</td>
<td>insert, delete, exact, similarity search</td>
<td>X-tree, TV-tree, M-tree</td>
<td>Multimedia databases, Medical databases</td>
</tr>
<tr>
<td>&gt;50</td>
<td>?</td>
<td>?</td>
<td>search engine, data mining</td>
</tr>
</tbody>
</table>

In almost all the application stated above, the databases are very large and consist of millions of data objects with a lot of dimensions. The R-tree or The R*-tree work perfectly at a space with low dimensionality but their performance decrease when the dimensions number is greater than 6. Therefore, new and appropriate indexing techniques are needed to query and provide an efficient access to these high-dimensional data. In order to cope with these important emerging applications, researchers and
experts developed some indexing techniques. The X-tree and the TV-tree are some of these indexing methods that I will present in the coming sections.

2. X-tree

2.1 Overview

The X-Tree (eXtended tree) is a variation of the R-Tree. Researchers developed the X-tree indexing technique to overcome the R-tree problems. The R-tree and its derivatives suffer from the “curse of dimensionality”, a phenomenon where performance drops as the number of dimensions increases. The major problem of the R-tree is the overlapping bounding box when the tree is in higher dimensions.

![Performance of the R-tree Depending on the Dimension (Real Data)](image)

Overlap of a node inside a R-tree is the percentage of the spaces that was covered by more than one hyperrectangle.

The X-tree is based mainly on the R*-tree which is more powerful than Rtree or R' -tree. The X-tree tries to avoid the overlaps generated by R*-tree in high-dimensional data. It used an extended directory called the supernode to solve the overlapping problem created by R*-tree. By doing so, it avoid a useless splits which may result in high overlapping of the child nodes. The X-tree tries always to balance and adjust between the linear and hierarchical structure without having high overlap of nodes.
2.2 X-tree structure

Three different nodes compose the X-tree:

**Data node:** contain minimum bounding rectangles (MBRs) and pointers to the actual data objects.

**Normal directory node:** contain MBRs with pointers to sub-MBRs. It also contains a split history used by inserting algorithm

\[
\text{MBR}_0 [\text{SplitHistory}_0] \rightarrow \text{MBR}_n [\text{SplitHistory}_n] \rightarrow \text{Ptr}_0 \rightarrow \ldots \rightarrow \text{Ptr}_n
\]

Structure of a Directory Node

**Supernode:** larger directory node of variable size created to avoid further splitting which means less overlapping. These supernodes are created only if there is no other possibility to avoid overlap. But in some cases, this option may be avoided by choosing an overlap-minimal split axis. This option will be discussed in the next section.

![Structure of the X-tree](image)

2.3 Algorithms

Insertion is the most important algorithm of the X-tree because when we have a good structured tree it will be easy to query (search or retrieve) and update the data objects into the tree. The insertion algorithm follows the following steps:

1. Determines the MBR in which to insert the data object and recursively calls the insertion algorithm to actually insert the data object into the corresponding node.

2. If no split occurs in the recursive insert, only the size of the corresponding MBRs has to be updated.

3. In case of a split of the subnode, an additional MBR has to be added to the current node, which might cause an overflow of the node. In this case, the current node calls the split algorithm.
4. The split algorithm tries to find a split of the node based on the topological and geometric properties of the MBRs.

5. If the topological split however results in high overlap, the split algorithm tries next to find an overlap-minimal split which can be determined based on the split history.

6. After the overlap-minimal split, if the nodes are underfilled, the number of MBRs will be compared against a threshold, if it does falls below the threshold then the split algorithm will be terminated without giving out a split. In this case, if the node is a normal directory node, it will be extended into a supernode. If it is already a supernode, then an extra block will extend it.

**Determining the overlap-minimal split:**

The split is called overlap-minimal iff \( \|\text{MBR}(S_1) - \text{MBR}(S_2)\| \) is minimal.

The split is called overlap-free iff \( \|\text{MBR}(S_1) - \text{MBR}(S_2)\| = 0 \).

### 2.4 Performance evaluation

This section is a comparison between the X-tree and R*-tree. The result here was obtained by using a constant size database with increasing dimensionality. The size of the database stays constant even when dimensions increase. Only the data is changed. The speedup at low dimensional over the R*-tree is about 30 for \( D \) (dimension)=8, and as high as 270 for \( D=16 \). The speedup was largely due to the fact that R*-tree has to access multiple paths in the directory at high dimensions.

![Speed-Up of X-tree over R*-tree on Point Queries](image)

Speed-Up of X-tree over R*-tree on Point Queries

*(100 MBytes of Synthetic Point Data)*

The high overlap in high dimensions forces the R*-tree to access most of the directory pages and use a considerable CPU time.
3. TV-tree

3.1 Overview

Almost in all multidimensional applications, such as medical image and multimedia databases, we need an expertise from the domain experts and specialists to provide us with necessary features that well describe objects in order to process some useful queries (exact match, range, nearest neighbor, all pair, sub-pattern matching). However, the number of features per object is between 10 and 100 in many modern applications, which cannot be handled efficiently by 2D and 3D spatial access method (R-tree, R*-tree, linear-quadtree, and so on) because as mentioned before these techniques suffer from dimensionality curse.

A new tree structure solves this problem by using a dynamic number of dimensions for indexing. It uses few dimensions to index nodes that are near to the root to discriminate among the objects. As it traverses the tree it use more and more dimensions. The tree is to contract and extend the feature vectors dynamically acting as a telescope. That is why researchers called this method Telescopic-Vector tree (TV-tree).

3.2 TV-tree structure

TV-tree organizes the data in a hierarchical structure. Objects (feature vectors) are clustered into leaf nodes of the tree, and the description of their Minimum Bounding Region (MBR) is stored in its parent node. Parent nodes are recursively grouped until the root is formed. The shape of the MBR can be chosen to fit the application; it may be a (hyper-) rectangle, cube, sphere etc. The simplest shape to represent is the sphere, requiring only the center and a radius. New features may be added when inserting new objects. This technique is used to decrease the phenomena of dimensionality curse.
**Node structure**

Each node represents the minimum-bounding region (MBR) of all its descendants. Each region has a center (telescopic vector) and a scalar radius. TMBR denotes the MBR with telescopic vector. TV-x denotes TV-tree with x active dimension. X determines the discriminatory power of the tree.

**Tree structure**

Each node contains a set of branches and each branch is represented by a TMBR, which contains all the descendants of that branch. TMBRs are allowed to overlap. Each node occupies exactly one disk page. Here are two examples of a TV-tree

![Structure of TV-1](image)

**TV-tree having multiple levels**

![TV-tree having multiple levels](image)

### 3.3 Algorithms

**Search**

For exact or range queries, the algorithm starts with the root and examines each branch that intersects the search region, recursively following these branches. Multiple branches may be traversed because TMBRs are allowed to overlap. Similarly, we can use branch-and-bound algorithm to do the
nearest-neighbor queries. Given a point, we compute the upper and lower bounds for the distance of branches descend the most promising one and disregarding branches that are too far away.

**Insertion**

To insert a new object, we traverse the tree, choosing the branch at each stage that seems most suitable to hold the new object. Insert the object in the reached leaf. The insertion operation may result to overflow that is handled by splitting the node, or by re-inserting some of its contents. After the insertion, split, or re-insert operations that might occur while adding new object, the TMBRs of the affected nodes along the path are updated.

The selection of the suitable branch follows the following criteria (in descending order)

1. **Minimum increase in overlapping regions within the node.**

   Choose the TMBR such that after update, the number of new pairs of overlapping TMBR is minimized within the node introduced. R1 is selected because extending R2 or R3 will lead to a new pair of overlapping regions.

2. **Minimum decrease in dimensionality.**

   Choose the TMBR with which the new object can agree on as many coordinates as possible, so that it can accommodate the new object by contracting its center as little as possible. R1 is selected over R2 because selecting R2 will result in a decrease in dimensionality of R2.
3. Minimum increase in radius.

R1 is selected over R2 because the resulting region will have a smaller radius.

4. Minimum distance from the center of the TMBR to the point.

R1 is selected over R2 because R1's center is closer to the point to be inserted.

The overflow caused by inserting or by extending a telescopic vector is handled by splitting the node, which is the most obvious way, or by re-inserting certain items to discard dissimilar items from a node and achieve better clustering. The following schema is followed to handle overflow:

- For a leaf node, a pre-determined percentage of the leaf contents will be reinserted if it is the first time a leaf node overflows during the current insertion (pick those that are farthest away from the center of the region). Otherwise, the leaf node is split in two.
- For an internal node, the node is always split; the split may propagate upwards.

**Deletion**

Deletion works as normal. In case there is an underflow, the remaining branches of the node are deleted and re-inserted. The underflow may propagate upwards.

**Extending and contracting**

As mentioned before the TV-tree is mainly based on *Extending and contracting* telescopic vectors. They are the most important aspect of the TV-tree algorithm.

Extending is done at the time of split and reinsertion and contraction occurs during insertion.
3.4 Experimental Results

The TV-tree outperforms the R*-tree in all operations. Here are the results represented in some figures:

<table>
<thead>
<tr>
<th>Dictionary size</th>
<th>Disk access per insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$R^*$-tree</td>
</tr>
<tr>
<td>4,000</td>
<td>5.25</td>
</tr>
<tr>
<td>8,000</td>
<td>5.51</td>
</tr>
<tr>
<td>12,000</td>
<td>6.19</td>
</tr>
<tr>
<td>16,000</td>
<td>6.50</td>
</tr>
</tbody>
</table>

Disk access per insertion - object size 100 bytes/ page size 4K

A 4K-page size is used and 100 bytes per object.
TV-tree requires fewer nodes. Since both indices have the same object size and the same number of leaf nodes. It reflects that TV-tree requires fewer internal nodes. It means that the non-leaf levels will require smaller buffer. This property can be significant when the buffer space is limited.

References:

1. Informix Decision Support Indexing for the Enterprise Data Warehouse
4. Wu Hai Liang, Hubert, Lam Man Lung, Lo Ming Fun, Yuen Chi Kei, Ng Chun Bong: A Survey on High Dimensional Spaces and Indexing