Outline

- Join indexes
  - Foreign column join index
  - Multitable join index
- Indexing techniques used for High-Dimensional data
  - The X-tree
  - the TV-tree
The join operations are highly used to execute SQL queries in almost all relational DBMSs. New approach called join indexing start to be adopted by 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.

There are two different kinds of join index:

- The foreign column join index
- The multitable join index
Join indexes

The foreign column join index

Let us suppose two tables CUSTOMER and SALES

![Diagram showing the join between CUSTOMER and SALES tables]

These tables are joined by the cid primary key of the CUSTOMER table with the foreign key cid of the SALES table.
Join indexes

The foreign column join index

we want to perform the following query:

```sql
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;
```

we can select a set of sales rows that have male customers

down

presented as a **bitmap**
Join indexes
The foreign column join index

Similarly, we can restrict the SALES rows to those that have female customers and create another bitmap.

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.
Join indexes
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.

We have the following star schema:
Join indexes
The multitable join index

Customer Dimension
- cid
- gender
- city
- state
- zip
- hobby

Sales Fact
- cid
- pid
- day
- dollar_sales
- dollar_cost
- unit_sales

Product Dimension
- pid
- SKU
- brand
- size
- weight
- package_type

Time Dimension
- day
- week
- month
- year
- holiday_flg
- weekday_flg
Join indexes
The multitable join index

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';
```
Join indexes
The multitable join index

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. 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.
Join indexes

foreign column join index vs multitable join index

The efficiency gain of an MTJ index over three FCJ indexes is not tremendously significant.

MTJ is inflexible, because if we want to use the MTJ we must know the common queries in our workload which is not the case in all the time.
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 (e.g. B-tree).

But in the past decade, more and more advanced applications emerged to the market and require manipulation of multidimensional data:

- Geographical Information System
- Medical databases
- Multimedia databases
## Indexing High Dimensional data

<table>
<thead>
<tr>
<th>Dimensionality</th>
<th>Operations</th>
<th>Techniques</th>
<th>Applications</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,, K-D-B-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>
Indexing High Dimensional data

The databases are very large and consist of millions of data objects with a lot of dimensions in almost all the applications mentioned.

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.
Indexing High Dimensional data

Therefore, new and appropriate indexing techniques are needed to query and provide an efficient access to these high-dimensional data.

The X-tree

TV-tree
The X-Tree (eXtended tree) is a variation of the R-Tree.

It was developed 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.
Indexing High Dimensional data
The X-tree

Performance of the R-tree Depending on the Dimension (Real Data)
Overlap of a node inside a R-tree is the percentage of the spaces that was covered by more than one hyperrectangle.

Overlap and Multi-Overlap of 2-dimensional data
The X-tree is based mainly on the R*-tree which is more powerful than R-tree or R+-tree.

It 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.

The X-tree tries always to balance and adjust between the linear and hierarchical structure without having high overlap of nodes.
Indexing High Dimensional data
The X-tree

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

\[
\begin{array}{c|c|c}
\text{MBR}_0 & \text{SplitHistory}_0 & \text{Ptr}_0 \\
\end{array}
\quad - \\
\begin{array}{c|c|c}
\text{MBR}_{n-1} & \text{SplitHistory}_{n-1} & \text{Ptr}_{n-1} \\
\end{array}
\]

Supernode: larger directory node of variable size created to avoid further splitting which means less overlapping
The 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.
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:
**STEP1:** 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.

**STEP2:** If no split occurs in the recursive insert, only the size of the corresponding MBRs has to be updated.

**STEP3:** 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*.

**STEP4:** The split algorithm tries to find a split of the node based on the topological and geometric properties of the MBRs.
**STEP5:** 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.

**STEP6:** 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 \( \| \text{MBR } (S_1) \cap \text{MBR } (S_2) \| \) is minimal.
- The split is called overlap-free iff \( \| \text{MBR } (S_1) \cap \text{MBR } (S_2) \| = 0 \).
Indexing High Dimensional data

The X-tree

Performance evaluation

The 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.

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

(100 MBytes of Synthetic Point Data)
The speedup was largely due to the fact that R*-tree has to access multiple paths in the directory at high dimensions.

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

A new tree structure solves dimensionality curse 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 uses 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).
TV-tree structure (hierarchical structure)

- Objects (feature vectors) are clustered into leaf nodes of the tree
- The description of their Minimum Bounding Region (MBR) is stored in its parent node.
- Parent nodes are recursively grouped until the root is formed.
Indexing High Dimensional data
The TV-tree

- The shape of the MBR can be chosen to fit the application; it may be:
  - a (hyper-) rectangle,
  - cube,
  - sphere

The simplest shape to represent is the sphere, requiring only the center and a radius.
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.
TV-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
Indexing High Dimensional data

The TV-tree

Structure of TV-1
Indexing High Dimensional data

The TV-tree

TV-tree having multiple levels
Indexing High Dimensional data
The TV-tree

Algorithms

- Search
- Insertion
- Deletion
- Extending and contracting
Indexing High Dimensional data
The TV-tree

**Search**

- **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.

- **nearest-neighbor queries:**
  - use branch-and-bound algorithm: 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

- 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 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.*
Indexing High Dimensional data
The TV-tree

- 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 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.
Indexing High Dimensional data
The TV-tree

*Extending and contracting*

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

- The TV-tree outperforms the R*-tree in all operations. Here are the results represented in some figures (A 4K-page size is used and 100 bytes per object):

<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>
Indexing High Dimensional data
The TV-tree

Experimental Results

Disk access of TV-tree and R*-tree (exact match queries)
Indexing High Dimensional data
The TV-tree

Experimental Results

Spaces usage of TV-tree and R*-tree
Indexing High Dimensional data

The R-tree

R-tree structure
Indexing High Dimensional data

The TV-tree

A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles. The R-tree is shown on the right.
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