INDEXING TECHNIQUE USING FOR TEMPORAL DATABASE

Mohd Shafry Mohd Rahim, Zahabidin Jupri, Daut Daman
Faculty of Science Computer and Information System
Universiti Teknologi Malaysia, 81310 Skudai, Johor.
Tel no: 07-5532422 Fax no: 07-5574908
E-mail: shafry@utm.my, zaha@utm.my, daut@utm.my

ABSTRACT

Despite the extensive research has been performed in temporal database area over the last twenty years, it still remains a great challenge in managing temporal database. Temporal database management systems (TDBMS) aim to make temporal data management easier and more familiar to users or applications. The concepts of storing unique attribute in the database is a fundamental, but when adding the temporal element raises some intriguing questions and new difficulties. Temporal database has to retrieve the past, present and future data. Therefore, the database becomes larger and more complex. Furthermore, the data structure, access method and query processing are to be considered. Recently, one relevant issue has captivated many researchers interest is low response time for data retrieval. Indexing technique is one approach that can improve the query performance and provide faster response time. Here, several indexing techniques for temporal database have been reviewed, also summarize their implementation and then compare against each other.

Keywords: Databases, Temporal Databases, Indexing, Data Retrieval, Data Management

1 INTRODUCTION

Database Management System (DBMS) were first developed to manage the current state of a database. However, in some applications, the past states of a database are also very important. Temporal database focuses on the management for any state of the data, e.g. past or current or future. In the temporal database, records are never physically deleted. Due to the large amount data in a temporal database, the data structure, access method and query processing are to be considered. This paper presents a comparison of proposed temporal access methods i.e. indexing techniques for temporal data. Here, several indexing techniques for temporal database have been reviewed, also summarize their implementation and then compare against each other.

A lot of research has been done on temporal databases and they have identified that classical indexing technique cannot handle huge amount of data in a database (Behl S, 2002; Samoladas V, 2001; Jiang L et al, 1999; Kouramajin V, 1994). Hence, some new or modification of previous technique has been proposed to overcome this issue. There are several features of index key must be considered to design and developing a new indexing technique (Ghanjaoui, 2003; Yannis M et al, 2002; Chortaras A, 2002). The index should be small and utilize space efficiently, also should support ad hoc and complex queries and speed up join operations. Besides that, the index should be easily dynamically generated and easy algorithms, implement and maintain, also efficiency for point and range queries as well as range queries.

2 TEMPORAL INDEXING TECHNIQUE

2.1 BINARY TREE (B-TREE)

 Nowadays, databases are increasingly being important for advanced application (O’Neil P and Quass D, 1996; Datta et al, 1998; Vanichayobon S and Gruenwald L, 1998; Kratky M et al, 2000; Chin Ooi B, 2001) such as object-oriented databases, geographic and spatial databases, temporal databases, data warehousing, high dimensional databases and XML databases. Deal with these applications, there are many specialized indexing techniques have been proposed and mostly, extension of B Tree index structure (A. Nascimento M and H. Eich M, 1995; Chin Ooi B, 2001). The B Tree is widely deployed in commercial systems and it is also easy to implement and understand.

Nowadays, databases are increasingly being important for advanced application (O’Neil P and Quass D, 1996; Datta et al, 1998; Vanichayobon S
and Gruenwald L, 1998; Kratky M et al, 2000; Chin Ooi B, 2001) such as object-oriented databases, geographic and spatial databases, temporal databases, data warehousing, high dimensional databases and XML databases. Deal with these applications, there are many specialized indexing techniques have been proposed and mostly, extension of B Tree index structure (A. Nascimento M and H. Eich M, 1995; Chin Ooi B, 2001). The B Tree is widely deployed in commercial systems and it is also easy to implement and understand.

Figure 1: The B Tree structure

Figure 1 shows an example of B Tree structure, which includes sets of 18 key values ranging from 6 to 88. However, conventional database structure always based on single version (Chang C. M and McNamee L, 1989; Becker B et al, 1996; Salzberg B and J.Tsotras V, 1997; Jiang L et al, 1999). It can only store the data that valid now, mostly updated information, but not the past information. The past information is very useful for reference or analysis in some application. For example, the employer of a company wants to conduct survey to investigate the employee flow in his company in the past year. This query cannot be answered if the database only keeps the mostly updated information.

2.2 MULTIVERSION BINARY TREE (MVBT)

Becker and friends (Becker B et al, 1996) proposed a Multiversion B Tree (MVBT) to overcome the deficiencies of B Tree indexing. As described above, B Tree structure always based on single version, therefore MVBT structure was proposed instead of single version. The main idea is to keep old data and perform what is called a version split whenever a node is filled up. Then, only the live data is copied to new nodes avoiding replicating old data. A new version created at each update (inserts or delete operation) i.e. the $i^{th}$ update creates version-$i$.

In the MVBT structure, leaf node of the tree consists of key, in_version, del_version and info as shown in Figure 2. The ‘key’ is an index of that item where the ‘info’ is the exact information of the record. The in_version and del_version indicate the lifespan of the record. The in_version contains the present version when a newly record created while the del_version represents as ‘*’ which means this record not be deleted (still alive). Also shown in Figure 2, the inner node of MVBT has router, in_version, del_version, info. The router is the separator key. The in_version and del_version indicate the range of version which the node will refer to the info contains the root location of the subtree.

<table>
<thead>
<tr>
<th>LEAF NODE</th>
<th>INNER NODE</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;key, in_version, del_version, info&gt;</code></td>
<td><code>&lt;router, in_version, del_version, info&gt;</code></td>
</tr>
<tr>
<td>A    <code>&lt;10,1,*,info&gt;</code></td>
<td>B    <code>&lt;10,18,*,A&gt;</code></td>
</tr>
<tr>
<td><code>&lt;15,3,info&gt;</code></td>
<td><code>&lt;15,18,*,B&gt;</code></td>
</tr>
<tr>
<td><code>&lt;25,1,info&gt;</code></td>
<td><code>&lt;30,21,*,C&gt;</code></td>
</tr>
<tr>
<td><code>&lt;30,1,6,info&gt;</code></td>
<td><code>&lt;55,14,25,F&gt;</code></td>
</tr>
<tr>
<td><code>&lt;55,1,4,info&gt;</code></td>
<td><code>&lt;70,18,*,F&gt;</code></td>
</tr>
</tbody>
</table>

Figure 2: The structure of MVBT

There are three standard operations in MVBT such as searching, inserting and deleting. Table 1 shows the description of each operation. Searching process is to exact match query (key, version) and range query (lowkey, highkey, version). Inserting process is to insert a record with given key and info component into the current version and deleting process is to delete the (unique) record with given key from the current version. The capacity of one block is $b$ that means one block can contains up to $b$ records. There are two conditions for the MVBT structure while insertion or deletion.

1. Weak version condition - Each block has zero items or at least $d$ present version entries where $b = k * d$ for some integer $k$ (if less than $d$, underflow occurs)

2. Strong version condition - Total number of present version entries in a block must be between $(1 + \varepsilon) * d$ and $(k - \varepsilon) * d$ after version split (overflow or underflow will occur)

2.3 UNIVERSAL BINARY TREE (UB TREE)

The Universal B Tree (Markl V, 1999; Mauch, 2002) is a new indexing technique to organize multidimensional data in databases. It also overcomes the deficiencies of the B Tree indexing technique by integrating new Multidimensional Access Methods. It combines the B Tree and Z-Curve which these combinations of query processing algorithm have proven many advantages concerning the performance on disk space and on response times in numerous application domains.
(Markl V and Bayer F, 2000; Mauch 2002). The UB Tree structure is different from the MVBT structure, where it can support valid or transaction time dimension but MVBT only support transaction time dimension. The use of Z-Curve is to convert coordinates into Z-address where it is usually used space-filling curve function. The space-filling curve is a function that projects all points of an n dimensional universe into one totally ordered dimension. Figure 3 describes the Z-Address transformation algorithm and example for a Z-address calculation. The standard operation in UB Tree structure: searching, inserting, deleting and sorting.

1. Searching - Point Query Algorithm All dimensions in the universe are restricted to a certain value that the search algorithm will transforms its coordinate into Z-address to find this point. Then, with this address UB Tree is traversed in order to find the Z-region [a,b] that contain Z(x). Z(x) is searched in the corresponding page that loaded from disk into memory. Range Query Algorithm The range query is a type of query in several dimension in the universe is restricted. Usually, for this type of query, data will spread over more than one region. So that, all regions that intersect the Query Box have to be loaded into main memory and will be process.

2. Inserting - Let say, to insert tuple x = (x1, x2...xn) with Z(x), a point query is done to determine the Z-region, then insert x, if the maximum page capacity is exceeded, the Z-Region has to be split into two regions [a, b][b+1,c]. Then, the objects contained in [a, c] are restricted to the new regions.

3. Deleting - Let say, to delete tuple x = (x1, x2...xn) with Z(x), if the number of objects in the page is too small after the x removal, the Z-region will merge with the succeeding Region.

4. Sorting - The Tetris algorithm is used to sort data in UB Tree where it is a special caching technique to minimize response time and cache requirements (Markl V and Bayer F, 2000; Mauch 2002).

As describes above, Z-address will be used to find the desired point for searching operation. Z-Region is defined by two addresses [a, b] and it is space covered by interval on the Z-Curve. The entire universe is subdivided into Z-Regions, each represented by one leaf node. It is sufficient to store the end point of the proceeding region. Each Z-region [a,b] is stored in one page disk, this page is denoted by page (a:b). Pages are limited to a certain capacity C, i.e. one page can only store C tuples. When creating an empty UB Tree the whole universe is covered by one Z-Region (one leaf in the corresponding B Tree). With data being inserted this Z-Region is split up into smaller regions so that the maximum page capacity is not exceeded.

```
Input: x : tuple
Output: ε : z-address of x
// n : number of dimensions
// k : number of segments
for i = 1 to k
    for j = 1 to n
        bit (i*n+j) of ε = bit k of x,i
    end for
end for
```

![Figure 3: The Z-Address Transformation Algorithm](image)

## 2.4 TIME INDEX TREE (TI)

Traditional indexing techniques are difficult to use for the properties of temporal database for the following reason (Kouramajin V, 1994). First, basically the search values in the time dimension are intervals rather than time points. The second reason is conventional indexing scheme cannot be used to define total ordering on the interval value, which the valid time intervals of object versions will overlap in arbitrary ways. Thirdly, in a temporal database past version are not deleted so update operations occur in an append-only mode. As time movement, this means that the search space keeps expanding.

Kouramajin V (1994) clarify an indexing technique namely, time index that considered the above discussed features proposed by Elmasri et al (1991). It can be used to retrieve versions of objects that are valid at a specific time point or during a specific interval. A time index consists of a collection of object versions, which defines over a temporal relational database (TRDB). Hence, TRDB = {e11, e12... e1k, ... e1l1,el2,...,elm}, where eij refers to version j of object e_i. There are many temporal search operations that are efficiently supported by the time index, the basic operations are point based and interval based. The following definition is an interval based operation which
retrieves all object versions whose time intervals intersect with the search interval. Given a search interval, \( I_s = [t_a, t_b] \), find the following set of object versions \( SV \):

\[ SV (I_s) = \{ e_{ij} \in TRDB \mid ([t_a, t_b) \cap I_s) \neq \emptyset \} \]

The time index will be defined over the valid time interval \((- \infty, + \infty)\). An indexing point is created at the time points where either a new interval is started, or the time point at which an interval terminates. Version pointer \( (p_{ij}) \) is a pointer to denote to object version \( e_{ij} \).

\[ SIP = \{ t \mid (\exists e_{ij} \in TRDB) \land (t = e_{ij}.ts) \lor (t = e_{ij}.te) \} \]

Table 1 depicts an example of temporal database consisting of an EMPLOYEE relation. This table will be used to illustrate the structure of time index. According to this table, there exist 9 indexing points in \( SIP \) for all employee versions, \( SIP = \{0, 2, 4, 6, 8, 10, 11, 12, \text{now}\} \). Basically, the time index structure is based on B Tree structure, but the difference is, each leaf node of time index structure consists of \([t, \text{bucket}_{ptr}]\), where \( \text{bucket}_{ptr} \) is a pointer to bucket \( B(t) \) containing version pointers.

1.1.1 Table 1: EMPLOYEE relation

<table>
<thead>
<tr>
<th>NAME</th>
<th>DEPT</th>
<th>VALID TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp1</td>
<td>A</td>
<td>[0, 4)</td>
</tr>
<tr>
<td>emp1</td>
<td>B</td>
<td>[4, now)</td>
</tr>
<tr>
<td>emp2</td>
<td>B</td>
<td>[0, 6)</td>
</tr>
<tr>
<td>emp3</td>
<td>C</td>
<td>[0, 8)</td>
</tr>
<tr>
<td>emp3</td>
<td>A</td>
<td>[8, 10)</td>
</tr>
<tr>
<td>emp4</td>
<td>C</td>
<td>[2, 4)</td>
</tr>
<tr>
<td>emp4</td>
<td>A</td>
<td>[8, now)</td>
</tr>
<tr>
<td>emp5</td>
<td>B</td>
<td>[10, now)</td>
</tr>
<tr>
<td>emp6</td>
<td>C</td>
<td>[12, now)</td>
</tr>
<tr>
<td>emp7</td>
<td>C</td>
<td>[11, now)</td>
</tr>
</tbody>
</table>

Table 1 shows a time index structure (order=3) consists of the \( SIP \) for the EMPLOYEE versions. Each internal node in the tree contains at most two search values and three pointers.

A bucket is maintained for storing all data records (or references to it) valid for that version. For each bucket has many version pointers and mostly will be repeated from the previous bucket. An incremental scheme is developed to reduce the redundancy of version pointers and to make the time index practical in terms of storage requirements. All open versions at a time point in \( SIP \) have a pointer if it is the first entry of a leaf node. This indexing point is also known as leading entry of a leaf node.

There are three pointers at each leading entry, \( t_1 \), which point to bucket:

1. \( SC (t_1) \) – Continuous (contains pointers to all object versions that were valid at the previous indexing point and continue to be valid at the current indexing point \( t_1 \))
2. \( SP (t_1) \) – Incremental plus (contains pointers to all object versions whose start time is \( t_1 \))
3. \( SM (t_1) \) – Incremental minus (contains pointers to all object versions whose interval end time is \( t_1 \))

The main drawback of the time index is that it requires a large amount of storage and suffers from degradation in update performance. Thus, the time index+ (Kouramajin et al, 1994), which extends the time index structure, overcomes these deficiencies by providing an efficient new storage model for partitioning logical buckets and new method for handling object versions with long and very long time intervals. There are three categories of object version: 1) Short Lived Version (SLV): An object version whose time interval is short. 2) Long Lived Version (LLV): An object version whose time interval is long. 3) Very Long Lived Version (VLLV): An object version whose time interval is very long.

Time index treats these three types of object versions, similarly, so that it requires huge amount of storage. Then, TI+ provides special techniques to handle object version of type LLV and VLLV, requires considerably less storage. It structure is similar to Time Index structure. The main difference between these two access structures is the way intermediate results are maintained. The time index requires a large amount of storage to keep \( SC \) buckets, whereas the TI+ controls the size of \( SC \) buckets. The time index+ handle the redundancy in \( SC \) buckets in two ways, controlling redundancy in version pointers to \( LLVs \) and controlling redundancy in version pointers to \( VLLVs \). With this controlling process, TI+ provides an improvement in search time and requires less storage than time index.

3 INDEXING TECHNIQUE COMPARISON STUDY

All these indexing techniques support temporal data, either valid time or transaction time dimension. However, none of these indexing
techniques allow more than one version valid at the same time. Thus, in the future, we plan to develop an indexing technique for bitemporal database, which it support valid time and transaction time dimension.

As describes in Table 4, Time index can be used to retrieve versions of objects that are valid at a specific time point or during a specific interval. However, it requires a large amount of storage and suffers from degradation in update performance. Thus, the time index”, which extends the time index structure, overcomes these deficiencies by provide a special function for handling object versions with long and very long time intervals.

The multiversion B-tree is quite effective which can keep the query time and storage space, but still lack to support valid and transaction time dimension. While, the UB Tree is the multidimensional access methods that overcome the conventional indexing technique, mostly have great disadvantages such as space utilization or algorithm complexity.

<table>
<thead>
<tr>
<th>UB Tree Structure</th>
<th>Cost</th>
<th>No. of Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input : x : tuple to store in the UB Tree</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Output : none</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1. ε = Z(x)</td>
<td>c1</td>
<td>1</td>
</tr>
<tr>
<td>2. find [α : γ] in the UB Tree, such that α ≤ ε ≤ γ</td>
<td>c2</td>
<td>1</td>
</tr>
<tr>
<td>3. retrieve page (α : γ)</td>
<td>c3</td>
<td>1</td>
</tr>
<tr>
<td>4. insert x into page (α : γ)</td>
<td>c4</td>
<td>1</td>
</tr>
<tr>
<td>5. while count (α : γ) &gt; C</td>
<td>c5</td>
<td>n</td>
</tr>
<tr>
<td>6. choose β [α : γ], so that ½ C – ε ≤ count (α : β) ≤ ½ C + ε</td>
<td>c6</td>
<td>n.1=n</td>
</tr>
<tr>
<td>7. split page (α : γ) into page (α : β) and page (β : γ)</td>
<td>c7</td>
<td>n.1=n</td>
</tr>
<tr>
<td>8. end while</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

T(n) = c1 + c2 + c3 + c4 + c5 + c6 + c7 + c8
= a(n)
= O(n)

4 CONCLUSION

From the evaluation that has done, time index and MVBT is the best algorithm can be implementing. However, from the description in the previous section, MVBT is more suitable for transaction database which it is support transaction time dimension only. Thus, time index algorithm will be chosen for future research, which it can handle huge amount data and support valid time dimension. As a result, we plan to provide an indexing technique that can handle valid time and transaction time dimension, also overcome the time index deficiencies that it requires large amount of storage.

5 ACKNOWLEDGEMENTS

This work has been done under research grant project E-Science Fund RMK 9 Vol Number 79233 which is sponsor by Ministry of Science Technology and Innovation (MOSTI). Thank for the Faculty of Computer Science and Information System (FSKSM) and Research Management Center (RMC) Universiti Teknologi Malaysia for the management of the research program and facilities.

References:
Techniques for Temporal Database”. Southern Methodist University, Dallas.


