JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 1, January – February 2006

An efficient multiversion access control in a Temporal Object Oriented Database

G. Arumugam, Department of Computer Science, Madurai Kamaraj University, Madurai, India
M. Thangaraj, Department of Computer Science, Madurai Kamaraj University, Madurai, India

Abstract

Many data base applications require the storage and manipulation of different versions of data objects. To satisfy the diverse needs of these applications, current database systems support versioning [Hulten94] at a very low level. The proposed model demonstrates that an application independent versioning can be supported significantly at higher level. The model uses signature patterns [Norvag99], hash table and B+ trees. The outcome of the analysis shows that the proposed model is performing well for all possible operations on versions.

1 INTRODUCTION

In an object oriented database system, the object model is often useful to store information about different aspects of an entity. Normally, this information is stored as versions of the corresponding objects. There are two broad categories of versions such as system level and application level. The system level versions are created and maintained by the database system. Such versions are useful for concurrency control [Bertino94], transaction support and redundancy in distributed databases. In contrast, application-level versions are created by applications for specific purposes. Examples of such versions include alternative designs for the object, previous states of the object, and so on.

In a temporal object database system[TODB] [Nervag01], it is usually assumed that most of the accesses will be to the current version of the objects in the database. In order to keep these accesses as efficient as possible, and benefit from object clustering, the database is partitioned. The current version objects are stored in the current database, and the historical versions are stored in the historical database. When an object is updated in a TODB, the current version is first moved to the historical database, before the new version is stored in-place in the current database. The OID needs to be updated every time an object is updated.

Cite this column as follows: G. Arumugam and M. Thangaraj: “An efficient multiversion access control in a Temporal Object Oriented Database”, in Journal of Object Technology, vol. 5, no. 1, January-February 2006, pp. 105-116, http://www.jot.fm/issues/issue_2006_01/article4

The scope of this paper is to study about application level versions. The users are able to access any subset of versions of an object or choose a version based on specified properties. In order to maintain different versions of an object for efficient query operation, an effective indexing structure is needed. In this paper, we present an index-based model that controls the access of versioned objects in the database.

The remainder of the paper is organized as follows: Section 2 is devoted to the issues relevant to the version management. In Section 3, we describe the architecture for version control. Section 4 shows our performance evaluation result. Finally, in Section 5 we present conclusion.

2 RELATED WORK

The notion of versioning can be modeled using pairs of types, each pair consisting of a generic type and version type. Each versioned entity has single associated generic object and zero or more associated version objects. The generic object contains the information common to all of its versions. All the versions of a generic object have the same scheme, but they differ only in the values of their attributes.

Versioning model [Sciore93]

EXTRA V is a model, contains key words, which are versioned. Attributes appearing after this keyword in a type declaration are versioned and those appearing before the keyword are unversioned. A version of a conceptual object denotes a previous or current state of the object.

Multi-version schemes [Lomet89]

In a multi version database systems, each write operation creates a new version whereas read operation selects one of the versions. The concurrency control scheme must ensure that the selection of the version to be read is done in a manner that ensures serializability.

Multi-version indexing [MVI] [Thangaraj03][Bober97]

It extends single-version indexing schemes to handle multi version data. It outlines four major approaches such as chaining, data page version selection, primary index version selection and all index version selection. These approaches will work with the on page catching method for storing prior versions and the B+ tree is used as a data structure for indexing.

3 PROPOSED MODEL

The multi version indexing schemes are considered different in the way they accomplish version selection. The selection mechanism locates the appropriate version of a tuple from a collection of existing versions. Earlier approaches use only B+ tree [Bober97] to manage the different versions. The proposed model, ATVIM, supports efficient handling of multiple versions of a single object.

The ATVIM is an integrated model supporting both inheritance hierarchy and aggregation hierarchy. The ATVIM model in Fig.1 is an improved version of the model proposed by Thangaraj al. [Thangaraj99][Thangaraj03].

The index model consists of two indexes namely primary and auxiliary indexes. Both indexes are using the data structures with the combination of signature patterns [Norvag99], hash tables and B+trees.

The primary index is indexed on the values of the attribute A_n. It associates with a value K of A_n, the set of OIDs of instances of all classes that have K as the value of the attribute A_n. The second index, called auxiliary index, has OIDs as indexing keys. This auxiliary index associates with the OID of an object O and the list of OIDs of the parents of O. The leaf-records in the primary index contain pointers to the leaf-node records in auxiliary index, and vice-versa.

Therefore, the primary index is inverted with respect to the values of attribute A_n and it is used for retrieval operations. The auxiliary index is inverted with respect to OIDs of instances of all classes (except for the class, root of the path, and for its subclasses) in the scope of a given path. Basically, the auxiliary index is used to determine all primary records where the OIDs of a given instance are stored in order to efficiently perform delete and insert operations.

The primary index consists of a hash table, signature patterns and B+ trees. Each bucket in the hash table has a signature pattern [Norvag99] and a pointer to a B+ tree. The signature is a bit pattern that has a pointer array of *n* bits which holds 1 or 0, while mapping, the bit corresponding to the object key is set as 1. During searching, the object availability can be determined with one comparison. Any effective hashing function can be used to hash the given key into the hash table and hence into the corresponding B+ tree.

The format of the non-leaf node has a structure similar to the traditional index structure on B+ tree. The leaf node in the primary B+ tree has different structure that contains the information about classid, record length, key value, class directory, and offset. The class directory consists of a set of classes that are having instances with the key value in the indexed attribute. The offset shows the position of the OID in the record. The remaining part of the node structure consists of the number of OIDs in each class and their versions, and the OID along with the pointer to the corresponding object in the auxiliary index.

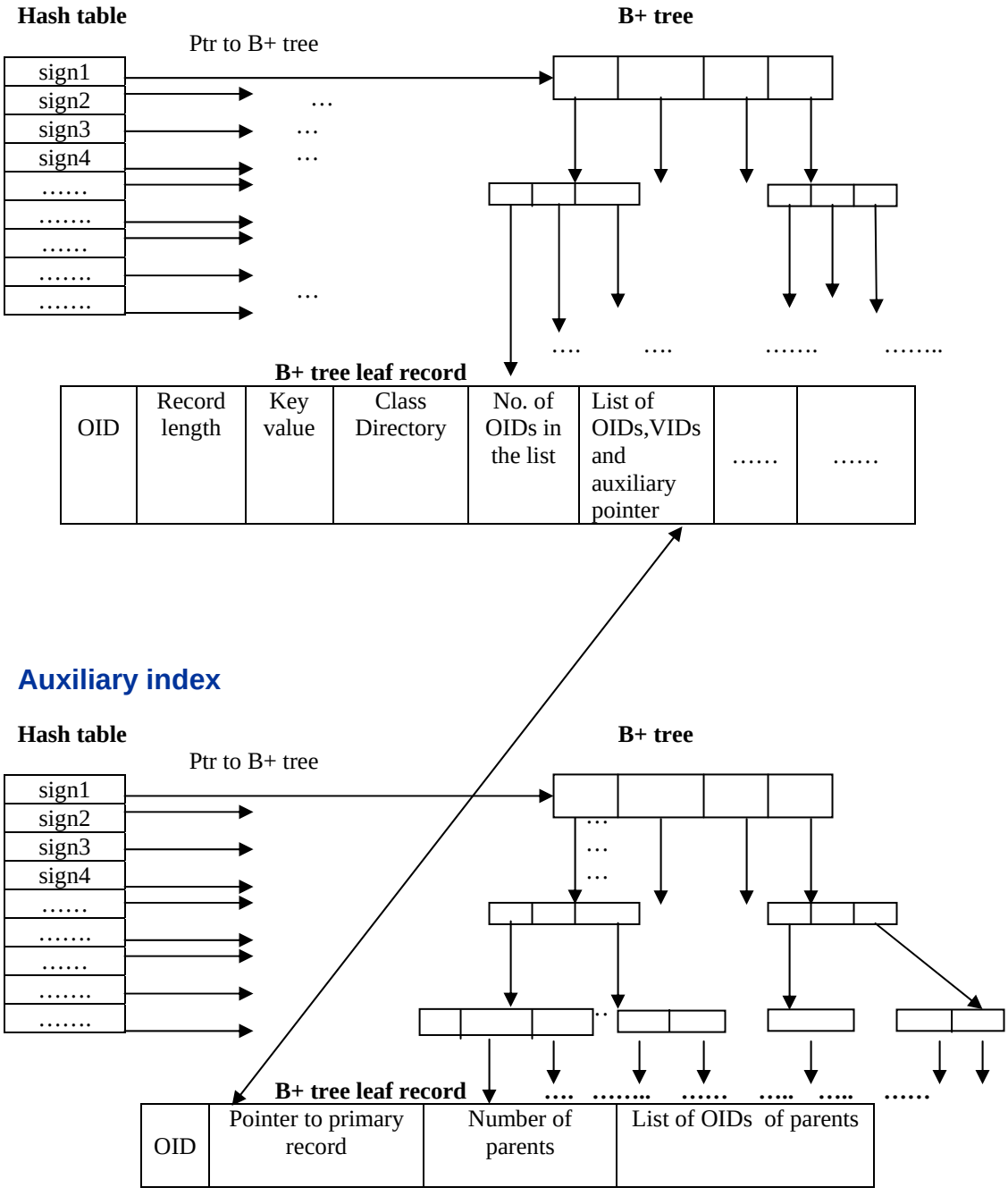
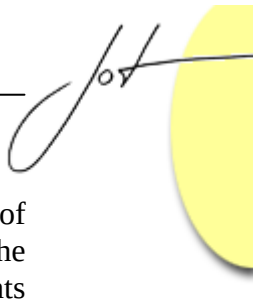


Fig. 1: ATVIM architecture

The auxiliary index also consists of a hash table, signature pattern and B+ trees. Every entry in the hash table has signature pattern that indicates the availability of the object and a pointer to a B+ tree. Any effective hashing technique can be used to hash the OID



of the object into the hash table and hence into the corresponding B+ tree. The format of a leaf node in the auxilliary B+ tree is as shown in Fig 1. It consists of the OID of the object, record length (number of OIDs referring to the particular object called as parents of the object), pointer to the primary record and list of parents.

Operations

In this subsection, we present an algorithm for retrieval, insert and delete operations for ATVIM model.

```
a)      Algorithm           : Retrieval
        Data structure      : See Fig. 1
        Input               : search value K
        Output              : List of OIDs
        Steps :
               Hash the key K, find the relevant bucket in primary hash table
               If the K is in Bit matrix
               {
               func find ( nodepointer, search value K)
               {
               if *nodepointer is a leaf,
                                 {
                                    find class directory and list of OIDs
               retrieve all versions using VID
               }.
               else
                      if K < K1 then return find(P0, K)

                      else

                             if  K >= Km  then return find ( Pm, K)
                                    //m = number of entries
                             else
                                    find I such that Ki <= K <= K i+1

                                    return find (Pi, K)
               end
               find the class directory in the  primary record node
               retrieve all the OIDs and its corresponding VIDS

b) Algorithm               : Insertion
       Data structure      : See Fig. 1
       Input               : Key  K
       Output              : Insert into proper position

       Steps :
               Hash the key K, find the relevant bucket in auxilliary hash table
               If the K is not in Bit matrix
               {
               set the corresponding bit in the matrix
               func insert (value V, pointer P)
                 {
               find the leafnode L that contains value V
               insert-entry(L,V,P)
               parentcount ++
               find the primary record using the pointer
               using class directory add new OID and increment OID,VID count
```

```
  }
func insert-entry(node L, value V, Pointer P)
{
If ( L has space for (V,P) then insert V,P in L
else     //split
{
       create node L'
       If (L is a leaf) then
              {
                     Let V' be the value such that [n/2] of
                     the values
                     L.K1,…,L.Kn-1, V < V'
                     Let m be the lowest value such L.Km >= V'
                     Move L.Pm,L.Km,…,L.Pn-1,L.Kn-1 to L'
                     If(V < V') then insert P,V in L
                     else insert P,V in L'
              }

       else
              {
                     Let V' be the value such that [n/2] of
                     the values
                     L.K1,…,L.Kn-1, V >= V'
                     Let m be the lowest value such L.Km >= V'
                     add Nil, L.Km,.., L.Pn-1, L.Kn-1,L.Pn to L'
                    delete  L.Km,..., L.Pm…,L.Kn-1,L.Pn from L
                     If(V < V') then insert V,P in L
                     else insert V,P in L'
                     delete Nil, V'from L'
              }
       if (L is not the root of the tree) then
                     insert-entry(parent(L), V', L')
       else   {
                     create new node R with child nodes L and L'
                     and the single value V'
                     make R as the root of the tree
              }
       if (L is the leaf node) then
                     set L'.Pn = L.Pn
                     set L.Pn = L'
              }
       }
}
```

c) Algorithm : **Deletion**
Data structure : See Fig. 1
Input : Key K
Output : Deletion of OID and its instance

Steps :

```
       Hash the key K, find the relevant bucket in auxilliary hash table
       If the K is  in Bit matrix {
       make the corresponding bit in the matrix as zero
       func delete (value V, pointer P)
         {
       find the auxiliary leafnodeL that contains value V
       delete-entry(L,V,P)
       parentcount --
       find the primary record using the pointer
```

```
            temporary list is formed to store the OIDs & VIDs
            remove all ids in the temporary list
            remove the pointer to the primary record
              }
            func delete-entry(node L, value V, Pointer P)
            {
            delete V,P from L
            If ( L is the root and L has only one remaining child ) then
            make the child of L as the new root of the tree and delete L
            else if(L has too few values/pointers) then
                  {
                   Let L' be the previous or next child of parent L
                   let V' be the value between pointers L and L' in parent L
                   If (entries in L and L' can fit in a single node) then
                   {
                      if ( L is the predecessor of L') then
                      swap_variables(L,L')
                      if (L is not leaf) then
                          append V' and all pointers and values in L to L'
                        else append all (Ki, Pi) pairs in L to L'
                          set L'.Pn = L.Pn
                          delete-entry(parent(L), V',L); delete node L
                       }
            else        // redistribution : borrow an entry from L'
                   {
                      If ( L' is a predecessor of L) then  {
                          Let m be such that L'.Pm is the last pointer in L'
                          Remove L'.Km-1,L'.Pm from L'
                          Insert L'.Pm,V' in L as a first value
                           by shifting appropriate pointers to the right
                          replace V' in parent(L) by L.Km-1 }

                   else  {
                          Let m be such that (L'.Pm, L'.Km ) is the last
                          pointer/value pair in L'
                          remove L'.Pm, L'.Km from L'
                          insert L'.Pm, L'.Km  in L as a first value
                          by shifting other pointers and values to the right
                          replace V' in parent(L) by L'.Km
                          delete Nil, V' from L'
                           }
                   }
                }
            }
      }
```

The earlier model, MVI uses the data structure B+ tree for the indexed access. The performance of the MVI model is fully depends on the height of the tree. As the height of the B+ tree is increasing with respect to the volume of objects, which increases the retrieval time of an object. But in our proposed model, each bucket in the hash table points to a B+ tree which reduces the height of the tree. As the keys are uniformly distributed among the trees, the height of the trees is reduced when compared to the MVI model. The searching for availability of the particular versions in the database can be easily identified with a single comparison using the signatures in each bucket.

4 PERFORMANCE ANALYSIS

In this section, we demonstrate the efficiency of ATVIM model with extensive experimental evaluation. This new model and earlier indexing model such as MVI are implemented using C++. The set of experiment explores the effects of data and query parameters on performance. We use synthetic datasets (100K-1M) that contain objects whose search key (ie. OID) are uniformly distributed in the data set. All experiments reported in this section were conducted on Pentium 4 2.x Ghz with 256MB RAM and 80 GB of secondary storage, running Windows 2000. In order to test the model and to find the performance, about 100000 objects were created and various queries were implemented on a particular index.

Storage Cost

In all experiments performed, we have obtained that the traditional index has the lowest storage cost. In particular, the MVI organization has less cost when compared to the ATVIM organization. However, the storage costs may not be crucial, since large capacity storage devices are today widely available for lower cost. Therefore, it may be preferable to design models that provide good performance, even if they have large storage requirements.

Retrieval cost

Let *h* is the height of the index, *np* is the number of pages that are to be accessed, *ap* is the probability of accessing another page if it is not available and *b* is hashing overhead and *l* is the version lookup cost. The cost of evaluating the query in ATVIM model is

$$C(\text{retrieve}) = \begin{cases} h+1+b+l & \text{if } as < ps \\ h+np+ap+b+l & \text{if } as > ps \end{cases}$$

Where *as* is the average size of a primary record and *ps* is a page size. The number of pages that are accessed (*np*) is a product of average number of pages required per record(*pp*) and number of classes (target of the query). The *ap* can be computed using *(pp –np)/pp*

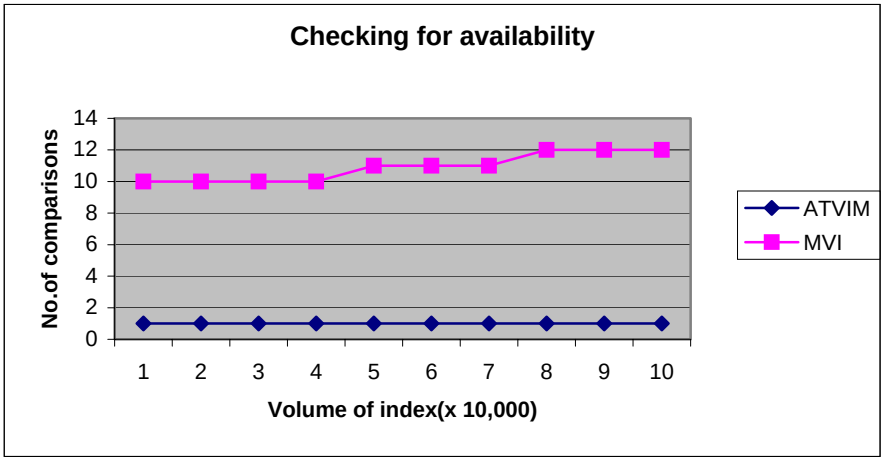
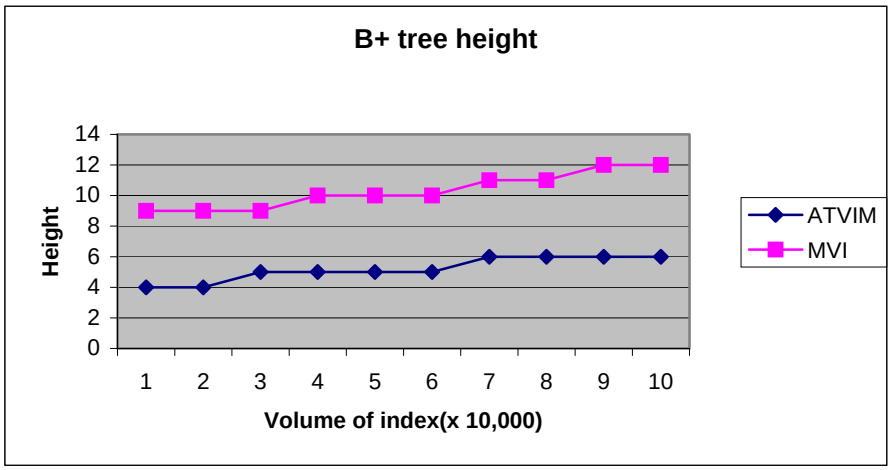


Fig.2: Availability checking

Fig. 2 shows the amount of comparison needed for checking the availability of the particular object. The ATVIM model needs just one comparison to know that availability of the particular object. But, the MVI model has to scan through the entire index structure



that consumes more time.

Fig. 3: Measuring the height

Fig. 3 illustrates the changes in the height of the tree across our given data set. The MVI model uses single B+ tree in which entire dataset has to be spread over. As the OIDs are uniformly distributed, the new model spreads the entire OIDs in various B+ trees in the structure that drastically reduces the height of the tree.

Fig. 4 compares the query performance of the two approaches. We applied a query that fetches the object versions from the same page. The time overheads were measured for retrieving the versions.

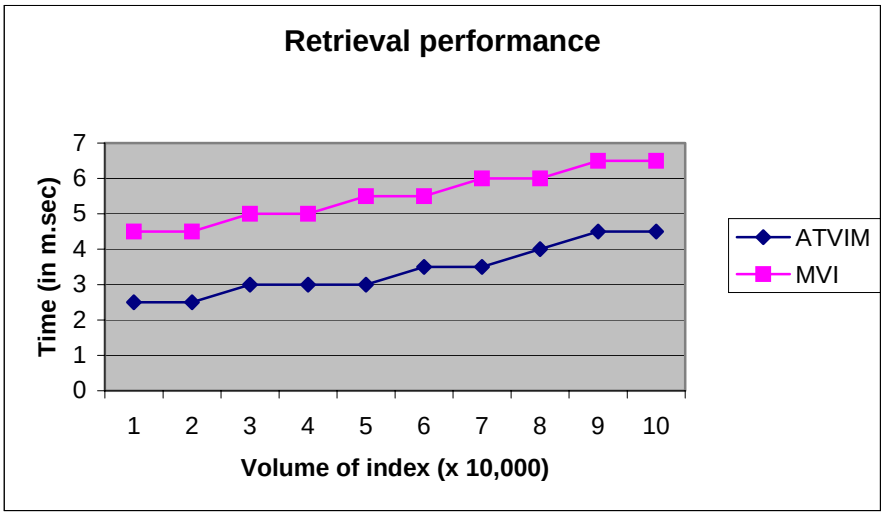
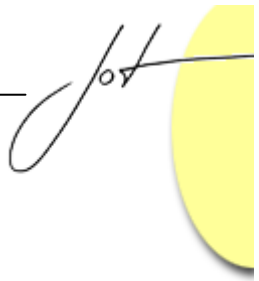


Fig. 4: Searching performance

We have presented the results of three experiments comparing the query performance of the alternative indexing schemes. One goal of these experiments was to determine the conditions under which placing version selection information in the table and the leaf node that reduces the query cost. On the positive side, such information can be used by a query to directly access an object version from its leaf index entry without having to go through one or more intermediate pages.

5 CONCLUSION

The model ATVIM proposed in this paper incorporates the aspects of signature patterns, B+ tree and hashing techniques to make efficient data retrieval. This model provides a flexible method to store and retrieve multiple versions. We conducted a simulation study of the two models and analyzed the results of this study. The result of this study indicates the importance of maintaining a hash table with signature patterns on index structures, as this turned out to be a key factor in determining the relative performance of the multiversion indexing schemes.



REFERENCES

[Bertino94] E Bertino:" A survey of indexing techniques for OODB", query processing for advanced data base systems,Morgan Kaufman publishers, inc, 1994.

[Bober97] Paul M.Bober, Michel J.Carey:"Indexing for multiversion locking: alternatives and Performance evaluation", IEEE trans on KDE, vol.9 no.1, Jan'97.

[Hulten94] Anders Bjornerstedt, Christer Hulten: "Version control in an Object Oriented architecture" Addison Wesley publishing Co., First ed.1994.

[Lomet89] David Lomet et.al.:"Access methods for multiversion data" ACM, pg 315-324,1989.

[Norvag01] Kjetil Nervag, "The Vagabond Temporal OID index: an index structure for OID indexing in TOODB",The VLDB Conference,2001.

[Norvag99] Kjetil Norvag: "Efficient use of Signatures in OODB" Proc. Of ADBIS'99,1999.

[Norvag98] Kjetil Norvag, K baratbergsengen: "Optimizing OID indexing cost in TOODB", Proc. Of FODO'98,1998.

[Sciore93] Edward Sciore : " Versioning and Configuration management in an Object Oriented data model ", The VLDB journal, June'93.

[Thangaraj99]M Thangaraj et.al.:" A new integrated indexing model for OODB", Proc. Of ROVIPIA, Malaysia, 1999.

[Thangaraj03]M Thangaraj, V R Kavitha: "An efficient multiversion control in Object Oriented data base systems", Proc. Of NACAC, Oct 2003.

About the authors

G Arumugam received his post-graduate degree in Applied mathematics from PSG College of Technology, Coimbatore and Ph.D degree from University of Piere and Marie curie, Paris, France in 1987. He is now the Professor and Head of Computer Science department at Madurai Kamaraj University, Madurai, TamilNadu, South India. He is an active researcher in databases, data mining, Bioinformatics and mobile computing and has published more than 50 papers in journals and conference proceedings. gurusamyarumgam@yahoo.co.in

M Thangaraj received his Post-graduate degree in Computer Science form Alagappa University, Karaikudi and M.Tech degree from Pondicherry, South India in 1998. He is now the research scholar cum senior faculty member in the computer science department at Madurai Kamaraj University, Madurai, Tamil Nadu, South India. His research area is databases and data mining and has published more than 20 papers in journals and conference proceedings. thangarajmku@yahoo.com