# Exploiting Versions for On-line Data Warehouse Maintenance in MOLAP Servers

Heum-Geun Kang and Chin-Wan Chung

Division of Computer Science
Dept. of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
{hgkang,chungcw}@islab.kaist.ac.kr

## Abstract

A data warehouse is an integrated database whose data is collected from several data sources, and supports on-line analytical processing (OLAP). Typically, a query to the data warehouse tends to be complex and involves a large volume of data. To keep the data at the warehouse consistent with the source data, changes to the data sources should be propagated to the data warehouse periodically. Because the propagation of the changes (maintenance) is batch processing, it takes long time. Since both query transactions and maintenance transactions are long and involve large volumes of data, traditional concurrency control mechanisms such as two-phase locking are not adequate for a data warehouse environment. We propose a multi-version concurrency control mechanism suited for data warehouses which use multi-dimensional OLAP (MOLAP) servers. We call the mechanism *multiversion concurrency control for data warehouses* (MVCC-DW). To our knowledge, our work is the first attempt to exploit versions for online data warehouse maintenance in a MOLAP environment. MVCC-DW guarantees the serializability of concurrent transactions. Transactions running under the mechanism do not block each other and do not need to place locks.

**Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002**

## 1 Introduction

The aim of a data warehouse is to enable users to make better and faster decisions [6]. To achieve this, the data warehouse collects information from several data sources and integrates them into a single database that contains historical, summarized and consolidated data. The data warehouse supports on-line analytical processing (OLAP) that provides tools for accessing and manipulating decision support information. The analysis is typically based on a multi-dimensional data model, known as a data cube [9].

A data cube is constructed from a subset of attributes in the database. Certain attributes are chosen to be measure attributes. Other attributes are selected as dimensions. The measure attributes are aggregated according to the dimensions. The data cube can be stored in the special data structures (e.g., array) or in the tables of the relational system. An analytical processing with data stored in special data structures is called multidimensional OLAP (MOLAP). Relational OLAP (ROLAP) is an analytical processing with data stored in a relational system.

As the data at the sources are changed, the data at the warehouse becomes out-of-date. The data at the warehouse should be maintained in order to provide users with up-to-date information. To make the data at the warehouse up-to-date, the changes to the data at the sources are propagated to the data warehouse.

Considering the performance, most of the commercial products gather changes to the source data and propagate the changes to the data at the warehouse periodically. Because the changes are gathered, the propagation of the changes becomes a large update processing. We call the processing a *maintenance transaction.* Since queries do not change the data at the warehouse, the maintenance transaction is the only transaction which updates the data at the warehouse.

OLAP systems are quite different from On-line Transaction Processing (OLTP) systems that execute a large number of relatively simple transactions. Since the aim of OLAP is to support the decision making, a query in OLAP systems tends to be complex and involves a large volume of data. Normally, users of OLAP systems execute a sequence of queries interactively, and the data at the warehouse must be consistent during the query transaction.

The differences between OLAP systems and OLTP systems can be enumerated as follows. The first difference is the transaction execution time. The length of query transactions executed in OLAP systems is very long. It can be of several minutes or hours. If some transactions are blocked by another transaction in the system due to the use of locking, the blocked transactions can be delayed for a long time. The second is that normally while several short update transactions are executed concurrently in the OLTP system, the number of update transactions (maintenance transactions) executed in the OLAP system is at most one at a time. The third is the volumes of data to be retrieved. Transactions in OLAP systems can access large portions of data.

By the features of OLAP transactions, if OLAP systems employ a typical locking mechanism or an optimistic mechanism for their concurrency control mechanisms, it induces both a high probability of conflicts and a high abort rate. The naive method commonly used in commercial OLAP systems is not to run queries during the maintenance time. This method propagates the changes at the data sources to the data warehouse without blocking. However, this method is often criticized as being too restrictive since a maintenance transaction and a query transaction cannot run simultaneously. As organizations become globalized, the OLAP systems should be able to respond to the queries submitted by users in multiple time zones [7, 8, 12, 16].

Some researches have been done to resolve the problem mentioned above in the ROLAP environment [10, 12, 15]. However, there is no literature which deals with the problem in the MOLAP environment. In this paper, we propose a multi-version concurrency control mechanism suited for data warehouses which use MOLAP servers that employ arrays for their storage structures. In fact, many of the leading OLAP companies use arrays in their multi-dimensional database systems for their basic storage structures [17]. We call our mechanism *multiversion concurrency control for data warehouses* (MVCC-DW). To our knowledge, our work is the first attempt to exploit versions to resolve the concurrency control problem of multi-dimensional arrays (MDAs) in the MOLAP environment.

**Contribution** In this paper, we propose MVCC-DW which is a new concurrency control mechanism for MOLAP servers. The MVCC-DW has the following features:

- **Non Blocking:** Both query transactions and maintenance transactions do not block each other and are not aborted, even though they use the same data items.
- **Serializability:** Our mechanism guarantees the serializability of both query and maintenance transactions, even though they run simultaneously.
- **No Lock:** Transactions do not need to place locks.

In addition, we implement our MVCC-DW prototype by modifying the Shore storage manager [5] and conduct an extensive experimental study with the dataset from the APB benchmark [1].

The remainder of the paper proceeds as follows. Section 2 introduces a typical MDA implementation as a background. Section 3 presents the basic idea, the data structures and the algorithms of MVCC-DW. Section 4 presents the experimental environment and the results. Section 5 surveys related works which exploit versions in ROLAP servers. Finally, Section 6 contains conclusions.

## 2 Multi-Dimensional Arrays for MOLAP

In this section, we describe a typical architecture of MDAs for MOLAP as a preliminary. The architecture shown in Figure 1 contains two B+-tree indexes, one for each dimension, an R*-tree index [2], and a chunked file.

The chunked file is a file structure that stores a multi-dimensional array. A large multi-dimensional array can be stored on disk in row-major or column-major order as a programming language manages arrays in memory. In this case, logically adjacent cells can be far apart on disk [17]. In a chunked file, a large multi-dimensional array is divided into a set of small multi-dimensional arrays (chunks) which is used as an access unit. A chunk contains a number of cells. A chunked file provides better access performance than a large array in row-major or column-major order [13].

Generally, MOLAP servers use a two-level indexing method [17] as shown in Figure 1. The B+-tree index, one for each dimension, maps dimension values in the dimension to array index values. In order to retrieve or update the value of a cell, we first find the chunk that contains the cell. The R*-tree index maps a sequence of index values to an object identifier (OID) of the object that stores the chunk that contains the cell. Leaf nodes in the R*-tree index have a set of entries, one for each chunk. An entry contains an OID and a *minimum bounding rectangle* (MBR) that is the boundary of a chunk.

A data cube requires a large disk space. Since the data cube is very sparse, we can compress the data cube by omitting the invalid cells of the data cube. We

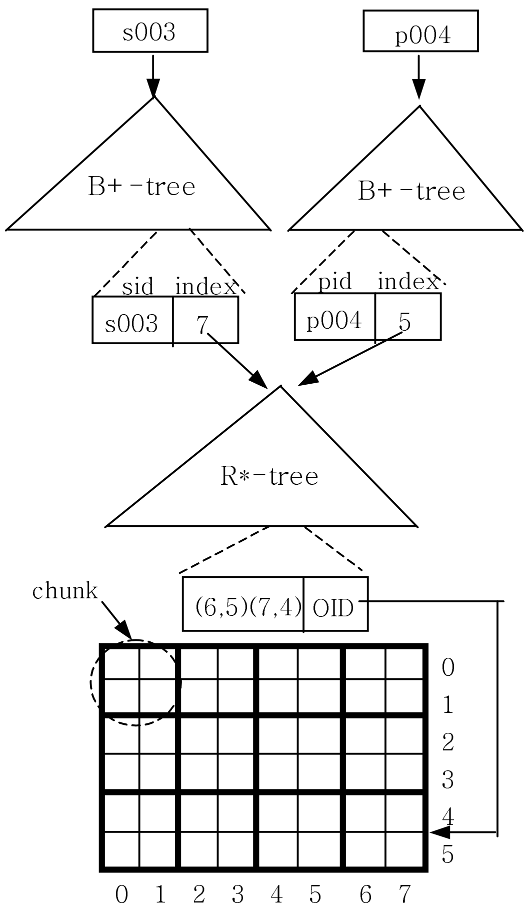

Figure 1: 2-Dimensional Arrays

use a form of compression, called "chunk-offset compression" [17]. To prevent waste of disk space, chunk-offset compression omits those invalid data cells and only stores a pair, (*offset*, *cellValue*), for each valid cell in each chunk instead of storing cells in row-major or column-major order. The value of *offset* is an offset from the beginning of the chunk to the cell in an uncompressed chunk. The value of *cellValue* is a value stored in the cell.

Figure 1 depicts an example for the retrieval of a cell value from a 2-dimensional array. The array in Figure 1 has 48 cells and 12 chunks. The size of a chunk is $2 \times 2$, so a chunk has 4 cells. The example explains how to retrieve the sales volume with dimension values of store identifier "s003" and product identifier "p004". We can get index values, 7 and 5, from the dimension values, "s003" and "p004", through the B+-tree indexes. Now, index values (7, 5) becomes the address of the cell that stores the sales volume. Then, we can obtain the OID of the object that stores the chunk that contains the cell through the R*-tree index. The R*-tree index in Figure 1 shows an entry stored in a leaf node. The entry contains index values (6,5) and index values (7,4), representing an MBR. Since the index values (7, 5) is contained in the MBR, we use the OID in the entry. Now, we can get the object that contains the cell, and get the value in the cell addressed by index values (7, 5) by computing the *offset* value with index value 7 and 5.

## 3 Multi-Versioned Multi-Dimensional Arrays

### 3.1 Motivation and Idea

As mentioned in Section 1, using a conventional locking mechanism results in a high blocking rate. Since query transactions and maintenance transactions tend to be long and complex, blocking could induce a significant delay. Also, optimistic concurrency control mechanisms are not adequate for the data warehouse environment because they can have a high abort rate. The high abort rate of long transactions induces a degradation in performance.

Our basic idea is to use a version mechanism. Since the maintenance in OLAP is done by the batch processing, a large unit of version control is adequate. Therefore, a chunk instead of a cell is used as the unit of version control. Typically, as mentioned earlier, a two-level indexing mechanism is used to retrieve and update the data in MOLAP servers. Thus, we devise a new index mechanism which supports the versioning concept. The technique we propose is inspired by the historical R-tree that was proposed as an access method for moving objects [11].

### 3.2 Revisions and Transactions

A query transaction should retrieve a consistent data set of the data warehouse. We use a revision as a consistent data with respect to query transactions. The state of a revision is either *frozen* or *active*. If a revision is frozen, the revision will not be updated any more. An active revision is one that is being updated and is the most recently created revision. The state of a revision can be changed only from active to frozen. While the number of frozen revisions is not limited, the number of active revisions is at most one at a time since we assume that the number of maintenance transactions (update transactions) is at most one at a time. We call the most recently frozen revision the *current revision* and call the least recently frozen revision the *oldest revision*.

Each revision is assigned a revision number, a number increased monotonically, when it is created. To prevent the waste of disk space, two revisions having adjacent revision numbers share chunks that are not changed between those two revisions. We call the revision number of the current revision the *current revision number*, call the revision number of the oldest revision the *oldest revision number*, and call the revision number of the active revision the *active revision number*.

A transaction in MOLAP servers is either a query transaction that is read-only or a maintenance transaction that updates a revision. A query transaction first selects the current revision to be used before executing a query, and then uses only the revision throughout its lifetime. This guarantees the serializability of concurrent transactions.

A maintenance transaction builds up a new revision. When the maintenance transaction starts, it first creates a new active revision, and then updates only the revision throughout its lifetime. . When the maintenance transaction finishes, it freeze the active revision.

Table 1 summarizes operations on a revision.

Table 1: Operations on a revision

| Operations | Who | When |
|---|---|---|
| Create the active revision | maintenance transaction | begin |
| Freeze the active revision | maintenance transaction | end |
| Open the current revision | query transaction | begin |
| Close a revision | query transaction | end |

### 3.3 The Architecture and Data Structures

As illustrated in Figure 2, the MVCC-DW architecture consists of two B+-tree indexes, a Multi-ReVision-tree (MRV-tree) index, and a chunked file. For each dimension of the MDA, a B+-tree index is used to map a dimension value to an index value. A sequence of index values, one for each dimension, addresses a cell in the MDA. Because the architecture uses a chunked file instead of a large array, the sequence of index values also specifies a chunk that contains the cell.

The MRV-tree index is used to map a sequence of index values to a chunk. The MRV-tree is composed of a sequence of RV*n*-trees (RV stands for ReVision and *n* represents a revision number). An RV*n*-tree index manages the mapping information from a sequence of index values to a chunk in revision *n*. The RV*n*-tree index for the current revision is called the *current* RV*n*-tree index. Similarly, the tree for the oldest revision is called the oldest RV*n*-tree index. If the revision *n* is frozen (active), the RV*n*-tree index is also frozen (active). Adjacent RV*n*-tree indexes share nodes if the nodes are not changed. This feature is similar to the HR-tree [11]. The chunked file stores chunks that are small multi-dimensional arrays. Each chunk has an OID, a chunk number, and a revision number. The OID specifies an object that holds the chunk. The chunk number specifies a chunk in a revision. If two chunks have the same chunk number and they belong to different revisions, one chunk is a version of the other. The revision number specifies a revision which was active when the chunk was created or versioned. Here, the change of an MDA is represented by a revision, while that of a chunk is represented by a version.

In the architecture shown in Figure 2, there are three RV*n*-tree indexes. The RV1-tree index, the RV2-tree index, and the RV3-tree index correspond to the revision 1, the revision 2, and the revision 3, respectively. The RV1-tree index has information about three chunks stored in the object 1, the object 2, and the object 3. The RV2-tree index manages information about three chunks stored in the object 2, the object 3, and the object 4. The RV1-tree and the RV2-tree share information about the object 2 and the object 3. This indicates that chunks stored in the object 2 and the object 3 are not changed during the maintenance transaction that builds up the revision 2. However, some cells in the chunk stored in the object 1 were changed during the maintenance transaction. Therefore, a new object was created, and the chunk stored in the object 1 was copied to the new object whose OID is 4.

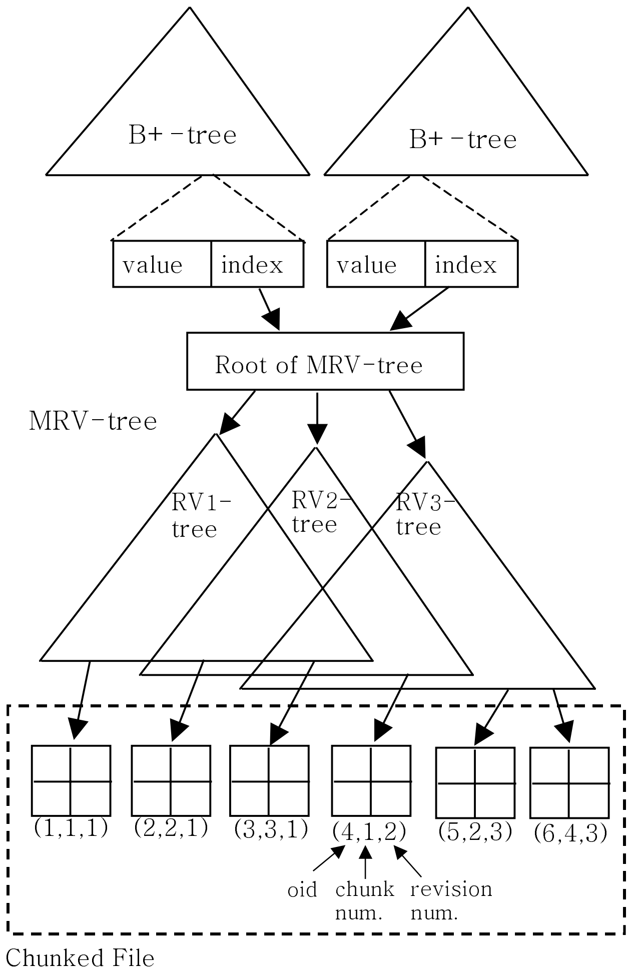

Figure 2: Multi Versioned 2-Dimensional Array

The RV2-tree index and the RV3-tree index have the same chunks stored in the object 3 and the object 4. The RV2-tree index has the chunk stored in the object 2 that is not owned by the RV3-tree index, and the RV3-tree index has the chunks stored in the object 5 and the object 6 that are not owned by the RV2-tree index. The chunk stored in the object 5 has the

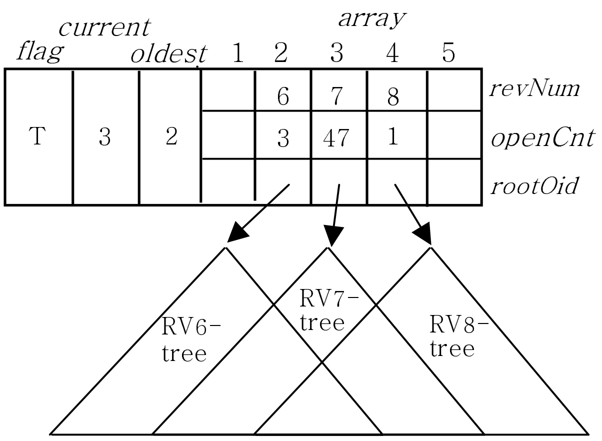

Figure 3: Root Node of the MRV-tree

same chunk number 2 as the chunk stored in the object 2, which indicates that the chunk in the object 5 is a version of the chunk stored in the object 2. The chunk stored in the object 6 has the chunk number 4 that no other chunk has. This indicates that the chunk is created during the maintenance transaction that builds up the revision 3.

Figure 3 shows the root node of the MRV-tree. The root node of the MVR-tree has information needed to manage several RV$n$-tree indexes. It contains a *flag* field, a *current* field, an *oldest* field, and an array of triple values. The *flag* field (T or F) indicates whether a maintenance transaction is running or not. The *current* field specifies the current revision. The *oldest* field indicates which RV$n$-tree index is used for the oldest revision for the garbage collection.

Each triple values, (*revNum*, *openCnt*, rootOid), in the array stores information for a revision. The *revNum* field denotes the revision number, the *openCnt* field is an open count that indicates the number of transactions that are accessing the revision, and the *rootOid* field stores an OID which represents an object that contains the root node of the RV$n$-tree corresponding to the revision. The *openCnt* field is used for the garbage collection. If the oldest revision is not the current revision and *openCnt* for the oldest revision is zero, then the oldest revision is garbage-collected.

In Figure 3, Value "3" in the *current* field means that the RV7-tree index which is denoted by the *revNum* 7 in the third entry of the array is the current RV$n$-tree index. Value"2" in the *oldest* field means that the RV6-tree index is the oldest RV$n$-tree index. The oldest RV$n$-tree index is a candidate for the garbage collection. Value "T" in the *flag* field means that a maintenance transaction is currently running.

The RV$n$-tree is a modified version of the R*-tree [2]. One difference between the RV$n$-tree and the R*-tree is that each node in the RV$n$-tree includes a field (*rNum*) for a revision number. A directory node (internal node) of the RV$n$-tree consists of $(rNum, L_1, L_2, ..., L_n)$. rNum stores the revision

| *rev-num* | *offset1* | *cellValue1* | *offset2* | *cellValue2* |
|---|---|---|---|---|
| 8 | 2345 | 7 | 3456 | 8 |

Figure 4: An Example of a Chunk

number of a revision that was active at node creation time. $L_i$ is the tuple for the $i$th child node $C_i$ and has the form of $(cp_i, MBR_i)$ where $cp_i$ is the address of $C_i$ and $MBR_i$ is the MBR enclosing all the entries in $C_i$. A leaf node in the RV$n$-tree contains $(rNum, D_1, D_2, ..., D_n)$. $D_i$ is the tuple for the $i$th data entry and has the form of $(P_i, MBR_i)$ where $P_i$ is an OID which indicates an object that contains a chunk and $MBR_i$ is the MBR enclosing all the cells in the chunk.

A chunk contains a revision number and an array of pairs, (*offset*, *cellValue*). An example of a chunk is depicted in Figure 4. Assume that the size of the chunk is $10 \times 10 \times 10 \times 10$, the revision number is 8, and the chunk has two valid cells. If the first valid cell has array indexes ( 12, 23, 34, 45 ) and value 7. The offset 2345 is computed by expression $((((((12\%10) \times 10) + (23\%10)) \times 10) + (34\%10)) \times 10) + (45\%10)$. The % symbol represents the remainder operator and value 10 represents a dimension size of the chunk. If the second valid cell has array indexes (23, 34, 45, 56) and value 8. The offset 3456 is computed by expression $((((((23\%10) \times 10) + (34\%10)) \times 10) + (45\%10)) \times 10) + (56\%10)$.

## 3.4 Algorithms

In this section, we explain the algorithms of the MV-MDA mechanism.

### 3.4.1 Algorithms for revisions

As mentioned earlier, when a maintenance transaction starts, it first creates a new active revision. Figure 5 shows the CreateRevision algorithm. Line 1 sets the *flag* field of the root node of the MRV-tree to "T" to indicate that a maintenance transaction is running. Lines 2-3 find the array index value for the current revision and the current revision number. Line 4 computes the revision number for the created revision. Then, the algorithm creates an RV$n$-tree index for the created revision. At first, the new RV$n$-tree shares all nodes but the root node of the current RV$n$-tree. The root node of the new RV$n$-tree has the same entries as the root node of the current RV$n$-tree. Next, the algorithm initialize the triple values, *revNum*, *openCnt*, and *rootOid*. The *revNum* field is initialized to the revision number of the created revision. The *openCnt* field is initialized to 0. The *rootOid* field stores the OID of the object that contains the root node of the new RV$n$-tree.

The active revision is frozen at the end of the maintenance transaction so that query transactions can re-

```
Algorithm CreateRevision()
input : rootOfMRV(root node of the MRV-tree )
Begin
1.  rootOfMRV.flag = "T"
2.  curIndex = rootOfMRV.current
3.  curRevNum = rootOfMRV.array[curIndex].revNum
4.  activeRevNum = curRevNum + 1 ;
5.  newRoot = new node
6.  Initialize rNum of newRoot to activeRevNum
7.  curRootOID = rootOfMRV.array[curIndex].rootOid
8.  oldRoot = the node stored in the object
        whose OID is curRootOID
9.  Copy the contents of oldRoot into newRoot
10. Initialize the triple values for the new revision.
11.     revNum = activeRevNum
12.     openCnt = 0
13.     rootOid = OID of newRoot
End.
```

Figure 5: The CreateRevision Algorithm

trieve the updated data in the revision. The FreezeRevision algorithm is described in Figure 6. To freeze the achieve revision, the algorithm updates the *current* and *flag* fields in the root node of the MRV-tree. The *current* field is updated to the index value of the array which corresponds to the active revision. The *flag* field is set to "F" which indicates that there is no running maintenance transaction.

```
Algorithm FreezeRevision()
input: rootOfMRV (root node of the MRV-tree)
Begin
1.  rootOfRMV.current = the index value of the array
2.      for the active revision.
3.  rootOfMRV.flag = "F"
End.
```

Figure 6: The FreezeRevision Algorithm

A transaction opens a revision at the beginning, uses the revision throughout its life time, and closes the revision at the end. Maintenance transactions open the active revision, and query transactions open the current revision. Figure 7 shows the OpenRevision algorithm that opens the current revision. The algorithm increases the value of the *openCnt* field by one, and returns the root node of the current RV*n*-tree. The value of the *openCnt* field indicates how many transactions open the revision, and it is used to check whether the revision can be garbage-collected. Then query transactions retrieve the values stored in the MDA through the root node as if the MDA is not versioned. The CloseRevision algorithm decreases the value of the *openCnt* field by one.

### 3.4.2 Insert Algorithm

In this section, we explain the Insert algorithm that is used by maintenance transactions. We omit some

```
Algorithm OpenRevision()
input: rootNodeOfMRV (root Node of the MRV-tree)
output: i (index value in the array of the root node
        of the MRV-tree)
output: rootNodeOfRV (the root node of the RVn-tree)
Begin
1.  Increment the openCnt field of the current
    revision by one
2.  Return the root node of the RVn-tree and the
    array index value corresponding to the current revision
End.
```

Figure 7: The OpenRevision Algorithm

detailed explanation for parts that are specified in the R*-tree.

```
Algorithm Insert()
Input: addr (address of a cell)
Input: val (value to be inserted)
Begin
1.  Find an appropriate chunk C1 that contains the cell
        whose address is equal to addr
2.  If found Then
3.      If rev-num of C1 =
            the active revision number Then
4.          Insert val into the cell specified by addr
5.      Else
6.          C2 = new chunk
7.          C2 = C1
8.          Initialize rev-num of C2
                to the active revision number
9.          Insert val into C2
10.         Call Propagate
11. Else
12.     C2 = new chunk
13.     Initialize rev-num of C2
            to the active revision number
14.     Insert val into C2
15.     Call Propagate
End.
```

Figure 8: The Insert Algorithm

The algorithm shown in Figure 8 considers two cases. The first case is that a chunk that contains the cell to be inserted already exists. If the chunk has the same revision number as the active revision, the Insert algorithm simply inserts the value into the cell, and returns. Lines 1-5 in Figure 8 process this. If the revision number of the chunk is different from that of the active revision, the chunk cannot be updated. Therefore, lines 6-10 create a new version of the chunk and insert the value into the new version. An entry for the new version of the chunk is inserted into the RV*n*-tree index so that transactions can find the chunk through the RV*n*-tree index. The Propagate algorithm explained below will do the job.

The remainder of the Insert algorithm processes the second case, in which there is no chunk that contains the cell. In that case, the algorithm creates a new chunk, and inserts the value into a cell in the chunk. An entry for the new chunk is inserted into the RV$n$-tree index so that transactions can locate the chunk through the RV$n$-tree index. The algorithm does this by calling the Propagate algorithm.

The Propagate algorithm shown in Figure 9 inserts an entry for a chunk into an RV$n$-tree index. The *newEntry* input parameter is the entry for the chunk to be inserted. If the chunk is versioned, the *oldEntry* stores the entry for the previous version of the chunk. If the chunk is created, the *oldEntry* stores null.

Lines 1-10 process the case in which the revision number of the RV$n$-tree node is equal to the active revision number. In that case, the node does not need to be versioned. When the value of *oldEntry* is not null, the algorithm simply exchanges the *oldEntry* with the *newEntry*. When the value of *oldEntry* is null, the value of *newEntry* is inserted into the *node*. If the *node* has rooms for an entry, the value of *newEntry* can be inserted. Otherwise, the algorithm calls the Overflow algorithm, and then it calls the Insert algorithm.

The next part of the algorithm processes the case in which the *node* needs to be versioned. In this case, a new node is created and the contents of the *node* are copied into the new node. The insertion is performed in the new node. If the *oldEntry* is not null, it should be removed from the new node because it is for the previous version of the node, *node*. When the new node has rooms for an entry, the entry for the *newEntry* is inserted into the new node and the Propagate algorithm is called again with the entry for the new node so that the update on the leaf node is propagated to the root node. When the new node has no rooms for a new entry, the algorithm calls the Insert algorithm after overflow processing is done.

The Overflow algorithm shown in Figure 10 deals with the overflow problem of a node. The algorithm is the same with that of the R*-tree.

The ReInsert algorithm shown in Figure 11 deals with the overflow problem. The algorithm removes some entries from the overflowed node, and inserts them into the RV$n$-tree. The algorithm sorts all the entries of the overflowed node using the method of the R*-tree to determine which entries are removed from the node and are re-inserted into the RV$n$-tree. If the revision number of the overflowed node is equal to the active revision number, some entries of the node can be removed and re-inserted. If not, the algorithm cannot modify the node. In that case, a new version of the node is created.

Lines 2-5 in Figure 11 treat the case that the revision number of the overflowed node is equal to that of the active revision. In this case, 30 % of the entries in the overflowed node are removed from the node. After some entries are removed, the bounding rectangle of the node is adjusted and it is propagated to the root node. Finally, the algorithm calls the Insert algorithm for each entry that is removed.

```
Algorithm Propagate()
Input: newEntry
Input: oldEntry
Input: node(node of the RVn-tree)
Begin
1.  If rNum of node =
          the active revision number Then
2.      If oldEntry ≠ null Then
3.          Remove oldEntry from node
4.          Insert newEntry into node
5.      Else
6.          If node has room for newEntry Then
7.              Insert newEntry into node
8.          Else
9.              Call Overflow
10.             Call Insert
11. Else
12.     newNode = make a new node
13.     Initialize rNum of newNode to
          the active revision number
14.     newNode = node
15.     If oldEntry ≠ null Then
16.         Remove oldEntry from newNode
17.     If newNode has room for newEntry Then
18.         Insert newEntry into newNode
19.         Call Propagate algorithm
20.     Else
21.         Call Overflow algorithm
22.         Call Insert algorithm
End.
```

Figure 9: The Propagate Algorithm

Lines 7-12 in Figure 11 deal with the case that the revision number of the overflowed node is not equal to that of the active revision. In this case, the algorithm should make a new version of the node and update the new version instead of the overflowed node. After the new version is created, all the entries in the overflowed node are copied into the new version. Then, the algorithm attaches the new version into the RV$n$-tree by calling the Propagate algorithm. 30 % of entries in the new version are removed, and the bounding rectangle of the new version is adjusted, and is propagated to the root node. The removed entries are re-inserted into the RV$n$-tree by calling the Insert algorithm for each entry.

The Split algorithm shown in Figure 12 makes a new node, distributes some entries in the overflowed node into the new node. The algorithm consists of two parts. The first part that is from line 1 to line 6 deals with the case that the revision number of the overflowed node is equal to that of the active revision. The remaining part treats the case that the revision number of the overflowed node is not equal to that of the active revision.

In case that two revision numbers are equal, a new node is created and all the entries in the overflowed

```
Algorithm Overflow()
Begin
1.  If the level is not the root level and this is the first
        call of Overflow in the given level during
        the insertion of one data Then
2.      Call ReInsert
3.  Else
4.      Call Split
End.
```

Figure 10: The Overflow Algorithm

```
Algorithm ReInsert()
input: oid (oid of a chunk)
input: thisNode
Begin
1.  Sort the entries of thisNode in
      descending order of their distances computed
      by the method of the R*-tree
2.  If rNum of thisNode =
        the active revision number Then
3.      Remove the first 30% of the entries from thisNode
4.      Adjust the bounding rectangle of thisNode
5.      Call Insert to insert the removed entries
6.  Else
7.      newNode = new node
8.      newNode = thisNode
9.      Initialize rNum of newNode
          to the active revision number
10.     Call Propagate
11.     Remove the first 30 % of the entries from newNode
12.     Adjusts the bounding rectangle of newNode
13.     Call Insert to insert the removed entries
End.
```

Figure 11: The ReInsert Algorithm

node are distributed into the overflowed node and the new node. After the distribution of entries, the bounding rectangle of the overflowed node is adjusted and the adjusted bounding rectangle is propagated to the root node. Then, the new node is attached into the RV$n$-tree by calling the Propagate algorithm.

In case that two revision numbers are not equal, two new nodes are created. All the entries in the overflowed node are distributed into two new nodes. After that, the new nodes are attached into the RV$n$-tree by calling the Propagate algorithm for each new node.

### 3.4.3 Garbage Collection Algorithm

In this section we explain the garbage collection algorithm which reclaims both the RV$n$-tree nodes and the chunks that are no longer used by any transactions. A query transaction opens the current revision, uses only the revision throughout its life time. The oldest revision that is not the current revision can be released for the space reallocation. However, the oldest revision can be used by some transactions that have opened the revision when it was current. We can release the oldest revision only when no transactions use it. Since the adjacent revisions share some tree nodes and chunks, a careful selection for the nodes and the chunks to be released is required.

```
Algorithm Split()
input: thisNode
Begin
1.  If rNum of thisNode =
        the active revision number Then
2.      Determine the axis to be splited
3.      newNode = new node
4.      Initialize rNum of newNode
          to the active revision number
5.      Distribute the entries in thisNode
          into thisNode and newNode
6.      Adjusts the bounding rectangle of thisNode
7.      Call Propagate
8.  Else
9.      newNode1 = new node
10.     newNide2 = new node
11.     Initialize rNum of newNode1
          to the active revision number
12.     Initialize rNum of newNode2
          to the active revision number
13.     Distribute the entries in thisNode into newNode1
          and newNode2
14.     Call Propagate for newNode1
15.     Call Propagate for newNode2
End.
```

Figure 12: The Split Algorithm

Figure 13 shows the GarbageCollection algorithm that reclaims the tree nodes and the chunks contained in the oldest revision except for those shared with other trees and revisions. The algorithm scans the oldest RV$n$-tree and the second oldest RV$n$-tree from the root nodes to leaf nodes. Lines 1-4 check to see whether the algorithm can proceed. The algorithm can proceed when there are at least two frozen revisions and no query transactions use the oldest revision.

Next, the algorithm makes two lists, *L1* and *L2*, stores the OIDs of the root nodes of the oldest RV$n$-tree and the second oldest RV$n$-tree into *L1* and *L2*, respectively. The algorithm repeats the steps between line 10 and line 16 for each level of the trees from the root level to the leaf level. For each level, the steps make a list of nodes belonging to the oldest RV$n$-tree but not belonging to the second oldest RV$n$-tree, and reclaim nodes in the list.

Finally, the algorithm makes a list of chunks belonging to the oldest revision but not belonging to the second oldest revision, and reclaims chunks in the list.

## 3.5 Correctness

In this section, we prove the correctness of MVCC-DW using the serializability theorem [4]. Let H be a history over transaction $T_1$, $\cdots$, transaction $T_n$. The serialization graph (SG) for H, denoted SG(H), is a

```
Algorithm GarbageCollection()
input: rootNode(root node of the MRV-tree)
Begin
1.  If there are some transactions that open
            the oldest revision Then
2.      return
3.  If the oldest revision is
            the same as the current revision Then
4.      return.
5.  oid1 = the OID of the root node of the oldest RVn-tree
6.  oid2 = the OID of the root node
                        of the second oldest RVn-tree
7.  L1 = a list containing oid1
8.  L2 = a list containing oid2
9.  For each level from the root level
            to the level 2 (just above of the leaf level)
10.     DiffList1 = L1 - L2
11.     DiffList2 = L2 - L1
12.     Read all the nodes whose OIDs are in DiffList1
          and write the OIDs stored in the nodes into List1
13.     Read all the nodes whose OIDs are in DiffList2
          and write the OIDs stored in the nodes into List2
14.     L1 = List1
15.     L2 = List2
16.     Reclaim all the nodes whose OIDs are in DiffList1
17. Process chunks
18.     ChunkList = L1 - L2
19.     Reclaim all the chunks whose OIDs are in ChunkList
End.
```

Figure 13: The GarbageCollection Algorithm

directed graph whose nodes are the transactions that are committed in H and whose edges are all $T_i \rightarrow T_j$ (i $\neq$ j) such that $T_j$ reads the value of a data item that $T_i$ wrote, or $T_j$ updates the value of a data item that $T_i$ read, or $T_j$ updates the value of a data item that $T_i$ wrote.

**Lemma 3.1** *Let $T_i$ use the revision $R_n$ and $T_j$ use the revision $R_m$ where n and m are the revision numbers of $R_n$ and $R_m$, and $n \neq m$. If there is an edge $T_i \rightarrow T_j$ in SG(H), then $n < m$.*

**Proof**) Suppose $n > m$. There are three cases. In the case that $T_i$ is a maintenance transaction and $T_j$ is a query transaction, $R_m$ does not contains the updates that $T_i$ performs. This contradicts $T_i \rightarrow T_j$. In the case that $T_i$ is a query transaction and $T_j$ is a maintenance transaction, when $T_j$ is running $R_n$ does not yet exists because $R_m$ is the most recently created revision. This contradicts $T_i \rightarrow T_j$. In the case that $T_i$ and $T_j$ are maintenance transactions, $T_i$ should create revision $R_n$ after $T_j$ freezes $R_m$. This contradicts $T_i \rightarrow T_j$. Thus, $n < m$. □

**Lemma 3.2** *If there is an edge $T_i \rightarrow T_j$ in SG(H) and both $T_i$ and $T_j$ use the same revision, then SG(H) should not contain $T_j \rightarrow T_i$.*

**Proof**) In the case that $T_i$ is a maintenance transaction and $T_j$ is a query transaction, $T_j$ should start after $T_i$ commits. This contradicts $T_j \rightarrow T_i$. The case that $T_i$ is a query transaction and $T_j$ is a maintenance transaction contradicts our assumption that a query transaction can open a revision frozen by a maintenance transaction. The case that $T_i$ and $T_j$ are maintenance transactions contradicts our assumption that a revision is built by one maintenance transaction. □

**Theorem 3.1** *An SG(H) for a history H produced by MVCC-DW is acyclic.*

**Proof**) by Lemma 3.1 and Lemma 3.2 □

Consequently, by Theorem 3.1 we can conclude that MVCC-DW produces serializable executions.

## 3.6 Clustering cells into chunks

The number of chunks to be versioned has a significant impact on the performance of MVCC-DW. The unit of the version control is a chunk. Although only a cell in a chunk is updated during a maintenance transaction, a new version of the chunk is created and all the cells in the chunk are copied into the new version. Therefore, it is important to cluster cells that are to be updated during a maintenance transaction into a small number of chunks.

### 3.6.1 Assigning index values

A dimension value must be translated into an index value (integer) before it is used as a component of the address of a cell. The mapping from a dimension value to an index value is maintained by a B+-tree index for a dimension. When assigning an index value for each dimension value, it is important that the index value should be unique for a dimension. However, it is irrelevant that which index value is assigned to which dimension value.

The dimension values for each dimension are hierarchically configured. For example, for the time dimension, there are some year values at the top level, and some quarter values belonging to a year value, and some month values belonging to a quarter value. All dimension values for a dimension are translated into index values in the same way, regardless of their levels.

The dimension values of the higher levels in the dimensional hierarchy represent summary information, and these values are more frequently updated than dimension values in the lower levels. If the values that are frequently updated are clustered into some chunks, the number of chunks to be versioned will be reduced. The clustering of the values in the higher levels can be achieved by allocating adjacent index values to the adjacent dimension values in the higher levels.

### 3.6.2 The shape of chunks

One of the methods to achieve good clustering of cells is to adjust the *shape* of chunks. The shape of chunks is determined by a sequence of lengths, one for each dimension. Assume that $X \times Y$ is the shape of 2-dimensional chunks. $X$ ($Y$) denotes the length of the first (second) dimension of the chunks. When a maintenance transaction is periodically executed, the maintenance transaction usually updates only the values of cells that have a specific value for the time dimension. For example, a maintenance transaction maintaining the sales volumes in January 15, 2002 does not update the sales volumes for other dates.

A chunk having cells addressed by several index values for the time dimension should be versioned several times. If all the cells in a chunk have only one value of the lowest level for the time dimension, the chunk does not need to be versioned. However, some chunks having summary information should be versioned since the summary information aggregates several data items of lower levels. We can reduce the number of chunks to be versioned using an appropriate shape of the chunks.

For example, assume that there is a two-dimensional array whose dimensions are the product and the time, the maintenance transaction is executed for each day, and there are two shapes of chunks, $5 \times 5$ and $25 \times 1$. Although both shapes contain the same number of cells, the number of chunks to be versioned during a maintenance transaction is quite different. A chunk whose shape is $5 \times 5$ contains sales volumes of 5 products for 5 days. In this case, the chunk is versioned each day for 5 days. A chunk whose shape is $25 \times 1$ contains sales volumes of 25 products for a day. In this case, the chunk is created during a maintenance transaction, and the value of any cell in the chunk is not changed during another maintenance transaction.

## 4 Experiments

In order to evaluate our mechanism, we implemented a prototype on top of the Shore storage manager [5] which provides the common database services such as B+-tree index, R*-tree index, transaction, concurrency control, recovery, and buffer management. We implement the RV*n*-tree by modifying the R*-tree code in the Shore storage manager. The prototype is based on a client-server architecture. Remote procedure calls are used for the communication between client processes and server processes. A server is multi-threaded, and a thread is invoked whenever a client is connected to the server.

In our experiments, we employed a dataset from the APB benchmark [1]. Figure 14 shows the schema of the dataset. The numbers in parentheses of Figure 14 represent the number of data items for each level of the dimensions. The dataset is composed of historical data and incremental data. A maintenance transaction adds the incremental data, which was a accumulated for the past one month, to the historical data. The size of historical data is 1,239,300, and that of incremental data is 72,900.

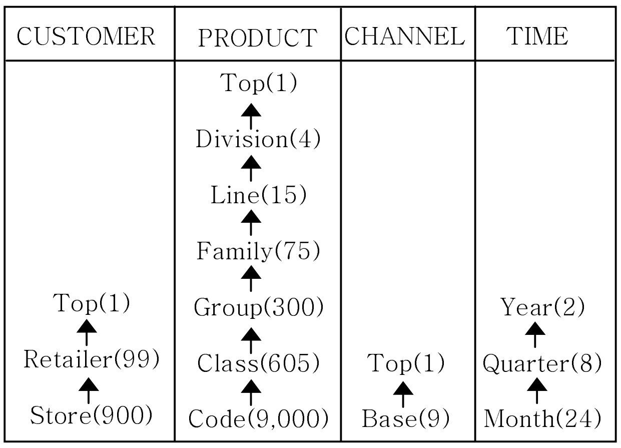

Figure 14: The hierarchical structure of the dimensions of the APB dataset

Table 2: Chunk Sizes

| | Customer | Product | Channel | Time |
|---|---|---|---|---|
| Size A | 10 | 10 | 10 | 1 |
| Size B | 20 | 20 | 20 | 1 |
| Size C | 30 | 30 | 30 | 1 |
| Size D | 40 | 40 | 40 | 1 |
| Size E | 50 | 50 | 50 | 1 |

All the experiments have been conducted on a Pentium computer with 500MHz CPU, 256M main memory and 10G hard disk. We first built up a data cube with the historical data, and the incremental data were loaded into the data cube through a maintenance transaction. We observed the status of the MVR-tree and the chunked file after the historical data were loaded and after the incremental data were loaded. The experiments were conducted with variable chunk sizes. Table 2 exhibits the several chunk sizes. For example, "Size A" is 10 for the customer, the product, and the channel dimensions, and 1 for the time dimension. "Size A" has 1000 cells. The reason why the size of the time dimension size is 1 is to reduce the number of chunks to be versioned during a maintenance transaction.

Table 3: Status of the MVR-tree and the chunked file after the historical data is loaded

| Chunk Size | RV0-tree Nodes | | | Num. of chunks | Valid cells in chunks | | |
|---|---|---|---|---|---|---|---|
| | 3 | 2 | 1 | | Min. | Max. | Avg. |
| A | 1 | 103 | 13,522 | 1,603,000 | 2 | 765 | 13 |
| B | 1 | 40 | 5,097 | 600,525 | 2 | 2,669 | 36 |
| C | 1 | 17 | 2,381 | 281,100 | 2 | 5,304 | 77 |
| D | 1 | 9 | 1,318 | 156,250 | 2 | 8,669 | 139 |
| E | 1 | 7 | 862 | 100,000 | 10 | 13,205 | 218 |

Table 4: Status of the MVR-tree and the chunked file after the incremental data is loaded

| Chunk Size | Num. of RV1-tree nodes | | | | Num. of Chunks | |
|---|---|---|---|---|---|---|
| | revision 0 | revision 1 | | | | |
| | 1 | 3 | 2 | 1 | revision 0 | revision 1 |
| A | 7,419 | 1 | 108 | 6,652 | 1,474,674 | 192,360 |
| B | 2,944 | 1 | 42 | 2,368 | 552,418 | 72,025 |
| C | 1,477 | 1 | 18 | 998 | 258,612 | 33,732 |
| D | 812 | 1 | 9 | 553 | 143,750 | 18,750 |
| E | 550 | 1 | 8 | 350 | 92,000 | 12,000 |

### 4.1 Experimental Results

Table 3 shows the status of the RV0-tree index and the chunked file after the historical data was loaded. The table exhibits the number of nodes of the RV0-tree for each level, the number of chunks created in the chunked file, the maximum number of valid cells in a chunk, the minimum number of valid cells in a chunk, and the average number of valid cells in a chunk. As the size of a chunk increases, the number of nodes decreases and the number of chunks also decreases. We can compute the approximate percentage of valid cells using the average number of valid cells in a chunk and the size of a chunk. When the chunk size is $10 \times 10 \times 10 \times 1$, the percentage of valid cells in a chunk is 1.3%. However, when the chunk size is $50 \times 50 \times 50 \times 1$, the percentage of valid cells in a chunk is 0.17%. Therefore, we can see that as the size of a chunk increases, the percentage of valid cells in a chunk decreases.

Table 4 shows the status of the RV1-tree index and the chunked file after the incremental data were loaded. The table presents the number of nodes of the RV1-tree index for each level and for each revision. In the table, the numbers of the RV1-tree's nodes whose revision numbers are 0 are not displayed, except the numbers for level 1. The reason is that there is no node that is shared in the RV0-tree in level 3 and level 2. Let's consider the case that the size of a chunk is $10 \times 10 \times 10 \times 1$. The RV1-tree has 14,071 leaf nodes, in which 7,419 nodes are shared with the RV0-tree and the remaining 6,652 nodes are created or versioned. The number of chunks in the chunked file is 1,667,034, in which 192,360 chunks were created or versioned. The number of chunks created or versioned during the incremental load is relatively small. This is very desirable because making a version of a chunk requires disk space and processing time.

We examined the execution time of query transactions that concurrently run with a maintenance transaction. To do this, we loaded the historical data, and then created one client process that executes a maintenance transaction loading the incremental data. While the maintenance transaction is executed as a thread, we created another client process that executes a query transaction. The aim of this experiment is to show that transactions do not block each other, and to examine the performance variation of query transactions among different chunk sizes. A query transaction in

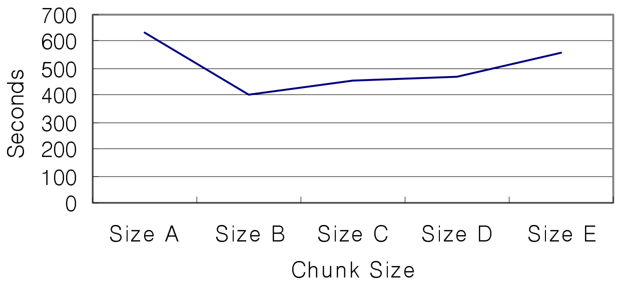

Figure 15: The performance of query transactions

the experiment executes 1,000 queries. A query retrieves 126 values (the combinations of levels in Figure 14) that consist of the value of a cell as well as all the values that aggregate the value of the cell and the values of other cells. A query transaction randomly selects cells to retrieve from the historical data. Figure 15 shows the performance result of the experiment. The result indicates reasonable response time when queries and a maintenance transaction are executed concurrently, and that the response time varies depending on the chunk size.

## 5 Related Work

While no technical study to exploit versions in MOLAP servers has been reported, some papers that deal with the issues in ROLAP environments have been published. [12] proposed a two-version no-locking mechanism (2VNL). The mechanism permits a query transaction to continue reading a view while the maintenance transaction is writing the new version of the view. To support two-version, the schema of a relation must be extended to include some attributes that store the old images. Queries issued by users are rewritten to fit into the extended schema. The number of versions is fixed to two. When the query transactions are too long or maintenance transactions occur frequently, two versions may not be enough and query transactions may be aborted.

In the DYVER mechanism proposed by [15], the number of versions is variable as needed rather than fixed. As 2VNL, DYBER requires the extended table schema. If a tuple is updated, a new version of the tuple is created and inserted into the table. Although one attribute is updated, all the other attributes are also copied. This leads to demand of a large disk space. As the number of tuples increases, the cost of query processing increases.

[10] presents another multi-version mechanism. In this mechanism, a maintenance transaction creates a new table for the future version, then it inserts new versions of tuples into the new table. Query transactions should search several tables to find an adequate version of the tuple.

## 6 Discussion and Conclusion

In this paper, we implemented the RV$n$-tree by modifying the R*-tree [2] of the Shore storage manager [5]. Thus, the RV$n$-tree inherits most of the properties of the R*-tree. The R*-tree is often criticized as having poor performance when the dimension of the tree is high [3]. The poor performance is due to the overlap of MBRs in directory nodes. The overlap of MBRs leads to several paths to visit when the tree is traversed from the root to a leaf. However, for the RV$n$-tree, the property can be avoided. The RV$n$-tree can be implemented with any other multi-dimensional access methods based on the tree architecture. In the past, there has been much research on the high-dimensional access methods, and many methods have been proposed. We can use them instead of the R*-tree.

We proposed a multi-version concurrency control mechanism, MVCC-DW that exploits versions for online data warehouse maintenance in MOLAP servers. MVCC-DW permits updates of the data at the warehouse while query transactions read the data. MVCC-DW has the following features. The first feature is that the MVCC-DW guarantees the serializable execution of both query and maintenance transactions running concurrently. The second is that both query and maintenance transactions do not block each other. The third is that all the transactions do not need to place locks. The fourth feature is that the number of versions of a data item is not limited to a fixed number, and is flexible as it is needed.

We developed a prototype that implements a multidimensional array with a chunked file and employs the MVCC-DW as its concurrency control mechanism. We conducted experiments with a dataset from the APB benchmark to check the practical feasibility of MVCC-DW. We found that query transactions run concurrently with a maintenance transaction without any conflicts, and the number of chunks that are versioned during a maintenance transaction is reasonably small.

## References

[1] OLAP Council, OLAP Council APB-1 OLAP Benchmark RII, *http://www.olapcouncil.org.*

[2] N. Beckmann and H. Kriegel, The R*-tree : An Efficient and Robust Access Method for Points and Rectangles, *Proc. 1990 SIGMOD*, pp. 322-331.

[3] S. Berchtold, D. A. Keim, and H. Kriegel, The X-tree : An Index Structure for High-Dimensional Data, *Proc. 1996 VLDB*, pp. 28-39.

[4] P. A. Bernsteinm, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[5] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling, Shoring up Persistent Applications, *Proc. 1994 SIGMOD*, pp. 383-394.

[6] S. Chaudhuri and U. Dayal, An Overview of Data Warehousing and OLAP Technology, *SIGMOD Record*, 26(1), 1997.

[7] M. Ester, J. Kohlhammer, and H. Kriegel, The DC-tree: A Fully Dynamic Index Structure for Data Warehouses, *Proc. 2000 ICDE*, pp. 379-388.

[8] H. Garcia-Molina, W. J. Labio, J. L. Wiener, and Y. Zhuge, Distributed and Parallel Computing Issues in Data Warehousing(Invited Talk), *Proc. of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[9] C. Ho, R. Agrawal, N. Megiddo, R. Srikant, Range Queries in OLAP Data Cubes, *Proc. 1997 SIGMOD*, pp. 73-88.

[10] S. Kulkarni and M. Mohania, Concurrent Maintenance of Views Using Multiple Versions, *Proc. of the International Database Engineering and Application Symposium*, pp. 254-258, 1999.

[11] M. A. Nascimento and J. R. O. Silva, Towards historical R-trees, *Proc. 1998 ACM symposium on Applied Computing*, pp. 235-240.

[12] D. Quass and J. Widom, On-Line Warehouse View Maintenance, *Proc. 1997 SIGMOD*, pp. 393-404.

[13] S. Sarawagi, M. Stonebraker, Efficient Organization of Large Multi-Dimensional Arrays, *Proc. 1994 ICDE*, pp. 328-336.

[14] T. Sellis, N. Roussopoulos, C. Faloutsos, The R+-tree: A Dynamic Index for Multi-dimensional Objects, *Proc. 1987 VLDB*, pp. 507-518.

[15] M. Teschke and A. Ulbrich, Concurrent Warehouse Maintenance without Compromising Session Consistency, *Proc. 1998 DEXA*, pp. 776-785.

[16] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom, A System Prototype for Warehouse View Maintenance, *Proc. of the ACM Workshop on Materialized Views: Techniques and Applications*, pp. 26-33, 1996.

[17] Y. Zhao, K. Ramasamy, K. Tufte, and J. F. Naughton, Array-Based Evaluation of Multi-Dimensional Queries in Object-Relational Database Systems, *Proc. 1998 ICDE*, pp. 241-249.