RC24288 (W0706-087) June 25, 2007
Computer Science

IBM Research Report

Efficient Bulk Deletes for Multi Dimensionally Clustered Tables in DB2

Bishwaranjan Bhattacharjee, Timothy Malkemus
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA

Sherman Lau, Sean Mckeough, Jo-anne Kirton,
Robin Von Boeschoten, John P. Kennedy
IBM Toronto Laboratories
Markham, Ontario
Canada

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Efficient Bulk Deletes for Multi Dimensional Clustered Tables in DB2

Bishwaranjan Bhattacharjee,
Timothy Malkemus
IBM T.J. Watson Research Center
Hawthorne, NY, USA
{bhatta, malkemus}
@us.ibm.com

Sherman Lau,
Sean McKeough,
Jo-anne Kirton,
IBM Toronto Laboratories
Markham, Ontario, Canada
{sherman, mckeough, kirton}
@ca.ibm.com

Robin Von Boeschoten,
John P Kennedy
IBM Toronto Laboratories
Markham, Ontario, Canada
{vboescho, kennedyj}
@ca.ibm.com

ABSTRACT

In data warehousing applications, the ability to efficiently delete large chunks of data from a table is very important. This feature is also known as Rollout or Bulk Deletes. Rollout is generally carried out periodically and is often done on more than one dimension or attribute. The ability to efficiently handle the updates of RID indexes while doing Rollouts is a well known problem for database engines and its solution is very important for data warehousing applications. DB2 UDB V8.1 introduced a new physical clustering scheme called Multi Dimensional Clustering (MDC) which allows users to cluster data in a table on multiple attributes or dimensions. This is very useful for query processing and maintenance activities including deletes. Subsequently, an enhancement was incorporated in DB2 UDB Viper 2 which allows for very efficient online rollout of data on dimensional boundaries even when there are a lot of secondary RID indexes defined on the table. This is done by the asynchronous updates of these RID indexes in the background while allowing the delete to commit and the table to be accessed. This paper details the design of MDC Rollout and the challenges that were encountered. It discusses some performance results which show order of magnitude improvements using it and the lessons learnt.

1. INTRODUCTION

Data warehouse sizes have been growing in leaps and bounds. An important concern is the storage costs associate with it. This is addressed by the periodic archiving of old data which might be accessed less often or by its summary removal from the database. Both methods require the mass delete of data from the warehouse. This is also known as Rollout or as Bulk Delete. The space thus freed up is used to make way for new data that is available. For example, a company might have a warehouse of 5 years of data. At the end of every month they might delete the oldest month of data and bring in data for the latest month.

In the past, such mass deletes were usually done in a maintenance window when the system load was low, like after midnight. Recent trends indicate users are moving towards a shorter time frame to perform this type of maintenance activities. Customers want their systems to be available almost 24 X 7 - even for a warehouse. Also, the amount of data being rolled out is becoming smaller but it is being done more frequently. These factors make an efficient online rollout mechanism very important for a database engine.

A key challenge in making an efficient online rollout mechanism is being able to handle the updates of RID indexes defined on the tables well. This is a well known problem and has been described in previous research work [1] [2]. RID indexes have pointers to records and have to be updated whenever the record they point to are deleted. A table might have many such RID indexes defined on them. Updating them entails significant locking, logging, index page IO as well as CPU consumption and has a strong influence on the response time of the delete as well as concurrency. This is especially true when the RID indexes are badly clustered and most of the index page IO ends up being synchronous due to bad locality of reference.

Another aspect of rollouts is that they might happen on more than one dimension. For example, one might want to rollout data based on shipdate at one time and orderdate on some other instance on the same table. One might want to remove data pertaining to a particular product or region etc. Also there might be further restrictions on these rollouts. For example, a user might ask to "delete orders older than 6 months provided they have been processed". The multi dimensionality of rollouts is thus an important characteristic and has to be addressed.

In DB2 UDB V8.1, a new data layout scheme called Multi Dimensional Clustering (MDC) [3] [4] [5] was introduced. This allows a table to be clustered on one or more orthogonal clustering attributes (or expressions). MDC initially supported a deletion capability based on the conventional delete mechanism of logging every row that was deleted and any indexes updated to reflect the delete. This works for mass deletes as well as single row deletes. Subsequently in DB2 UDB V8.2.2 Saturn [2], an enhancement called "Immediate Rollout" was incorporated, which allowed a user to more efficiently purge data from a table on dimensional boundaries by writing one log record for an entire block of data being deleted rather than one for every record. This technique greatly helps reduce logging requirements. It also

improves the response time of the deletes when there only dimensional block indexes defined on the table. However when there are badly clustered RID indexes defined, the cost of updating these indexes while doing the delete completely dominates the total cost of the delete. Thus in these cases, the Saturn enhancement does not impact the response time of the delete much.

In this paper we detail the design of a major enhancement to MDC deletes called "Deferred Rollout", which was incorporated into DB2 UDB Viper 2. This facilitates very efficient bulk deletes of data even when one has a lot of badly clustered RID indexes defined on the table. This is done by the asynchronous updates of these RID indexes in the background while allowing the delete to commit and the table to be accessed. We discuss some of the key challenges encountered in the design and the lessons learnt. We also discuss a performance study of MDC rollout which shows an order of magnitude gain in response time and compares its characteristics against conventional delete mechanisms.

The rest of the paper is structured as follows. Section 2 describes the MDC feature introduced in DB2 UDB V8, Section 3 describes how MDC deletes and bulk deletes work, Section 4 compares this against other bulk delete mechanisms and related work, and Section 5 gives a high level overview of the MDC Deferred Rollout mechanism incorporated in DB2 UDB Viper 2. Section 6 and 7 cover two important aspects of the Deferred Rollout - the ROBB and the AIC - in detail. In Section 8 we discuss the performance results of MDC Rollout and delete and compare it against non MDC delete. We conclude in Section 9 after a discussion of the lessons learnt.

2. MULTI DIMENSIONAL CLUSTERING IN DB2

Multi Dimensional Clustering (MDC) in DB2 UDB V8.1, allows a user to physically cluster records in a table on multiple orthogonal attributes or dimensions. The dimensions are specified in an ORGANIZE BY DIMENSIONS clause on a create table statement. For example, the following DDL describes a Sales table organized by region, year(orderDate) and itemId.

```
CREATE TABLE Sales(
date orderDate,
int    region,
int    itemId,
float price,
int yearOd generated always as year(orderDate))
ORGANIZE BY DIMENSIONS (region, yearOd, itemId)
```

Each of these dimensions may consist of one or more columns, similar to index keys. These could be base columns (like orderDate) or generated columns (like yearOd). In fact, a 'dimension block index' will be automatically created for each of the dimensions specified and will be used to quickly and efficiently access data. A composite block index will also be created automatically if necessary, containing all dimension key columns, and will be used to maintain the clustering of data over insert and update activity. For single dimensional tables since the dimension block index and composite block index will turn out to be identical, only one block index is automatically created and used for all purposes.

In our example, a dimension block index is created on each of the region, year(orderDate) and itemId attributes. An additional composite block index will be created on (region, yearOd, itemId). Each block index is structured in the same manner as a traditional B+ tree index except that at the leaf level the keys point to a block identifier (BID) instead of a record identifier (RID). A block is collection of pages. Currently block size is tied to the extent size of the tablespace on which the table is defined. Since each block contains potentially many records, the block indexes are much smaller than a corresponding RID index on a non MDC table. In one instance, a block index was of 71 pages and 2 levels whereas a corresponding RID index for a non MDC table was of 222,054 pages and 4 levels [4].

Figure 1 illustrates these concepts. It depicts an MDC table clustered on the dimensions year(orderDate), region and itemID. The figure shows a simple logical cube with only two values for each dimension attribute. Logical cells are represented by sub-cubes in the figure and blocks by shaded ovals. They are numbered according to the logical order of allocated blocks in the table. We show only a few blocks of data for a cell identified by the dimension values (1997,Canada, 2). A table (and a cell) can have upto 2^31 blocks. Note that a cell without any records will not have any physical representation in the table.

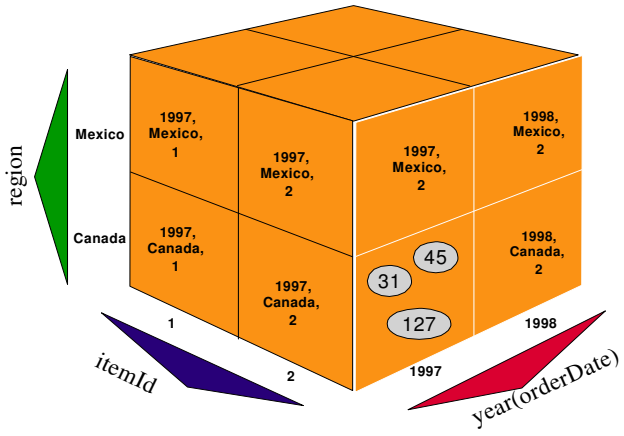

Figure 1: Logical view within a MDC table

A slice, or the set of blocks containing pages with all records having a particular key value as a dimension, will be represented in the associated dimension block index by a BID list for that key value. The following diagram illustrates slices of blocks for specific values of region and itemId dimensions, respectively.

In the example above, to find the slice containing all records with 'Canada' for the region dimension, we would look up this key value in the region dimension block index and find a key as shown in Figure 2(a). This key points to the exact set of BIDs for the particular value.

The DB2 UDB implementation was chosen by its designers for its ability to co-exist with other database features such as row-based indexes (a.k.a RID indexes), table constraints, materialized query tables, high-speed load, mass delete, hash partitioned MPP as well as an SMP environment.

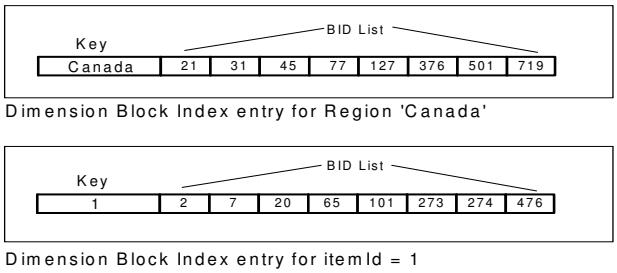

Figure 2: Block Index Key entries

MDC also introduced the concept of a Block Lock. The Block Lock is a locking mechanism which is between the Table Lock and a Record Lock in granularity. It allows for a block to be locked in various modes. Block Locks could escalate to Table Locks just like Record Locks do. However escalation of Record Locks to Block Locks is not currently supported.

Another data structure introduced in MDC was the Block Map. This stores information on the state of the blocks in a table. The information includes if the block is free, if it has been recently loaded, if it is a system block, requires Constraint enforcement etc. This information is used, among other things, during inserts and loads to select blocks to insert/load into. Figure 3 shows an example blockmap for a table. Element 0 in the block map represents block 0 in the MDC table. Its availability status is 'U', indicating that it is in use. However, it is a special block and does not contain any user records. Blocks 2, 3, 9,10,13,14 and 17 are not being used in the table and are considered 'F' or free in the block map. Blocks 7 and 18 have recently been loaded into the table. Block 12 was previously loaded and requires constraint checking to be performed on it.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
U	U	F	F	U	U	U	L	U	F	F	U	C	F	F	U	U	F	L	....

Figure 3: Block Map entries

A MDC dimension block index can be ANDed and ORed with other dimension block indexes as well as any record based index defined on the table. A full description of how they can be combined can be found in [3], [4].

3. MDC DELETES

A delete of a record of an MDC table, entailed logging of the entire record and updating any record indexes defined on the table. The record index updates were logged too. The freed up space is available for reuse by the same unit of work even before the delete commits. After the commit, all transactions are free to reuse the space. If the delete ended up emptying the block in which the record resided, then the dimension block indexes were updated and logged. Thus a dimension block index is updated very few times compared to a corresponding record index on a similar non MDC table delete in DB2. This has a positive impact on response time of the delete and amount of logging needed.

In DB2 UDB V8.2.2 Saturn, a feature named Immediate Rollout was introduced which allows for a more efficient delete of data along cell boundaries for MDC tables. It builds on the good points of MDC delete and also is submitted via a conventional SQL Data Manipulation Language (DML) delete statements. Thus users don't have to change their applications to tap this new feature. The compiler, under the covers, decides if the delete statement can be executed using this bulk delete mechanism. If it can be, then it generates a plan for its execution, else it switches to conventional MDC delete for that statement.

Using this feature, multiple, full cells can be deleted in any combination as long as it can be described using delete DML statements. Figure 4 shows the result of 4 different deletes on the MDC table described in Figure 1. They depict the result of purging the table of individual cells to entire slices of data. While the rollout is executing, concurrent access to the table is permitted provided lock escalation to the table level has not occurred. The rollout itself acquires an intent exclusive Table Lock, and exclusive Block Locks on blocks being rolled out. It does not get any individual Record Locks on records being deleted. Thus the chances of lock escalation due to this type of delete are much reduced compared to non MDC and this has a positive impact on the concurrent access of the table when large rollouts occur.

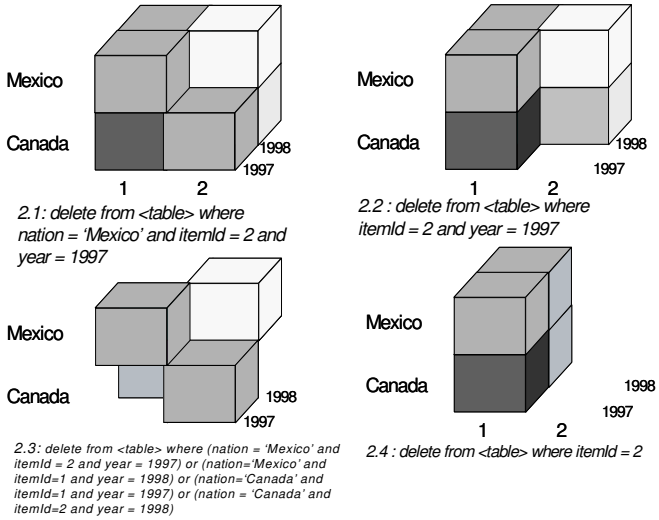

Figure 4: Example of rollout in a MDC table

For this type of delete, no record level logging is done as in conventional MDC delete. Instead, for all the records in the page, a single small log record is written. This indicates to the system that all records in the page have been deleted but the contents of the records themselves are not logged. Further, meta information stored in the page as well as the first page of the block is updated to indicate all records have been deleted and thus the pages of the block are free. This change is also logged.

This type of delete tends to process a block at a time as described above. When a block is rolled out, its corresponding entry in the blockmap is marked rolled out and the InUse bit is reset. This indicates that this block cannot be reused by the same transaction until the rollout is committed. All the Dimension Block Indexes are updated to reflect the fact that the block is no longer associated with its cell. It is to be noted that the block is still associated with the table after a commit and is reusable for any cell. It can be delinked from the table and returned to the tablespace by a table reorg.

Any row based indexes defined on the table are updated one row at a time. For each row, its entry in all the RID indexes is removed and this change is logged. Unfortunately in a large segment of real

customer scenarios, one can expect a lot of RID indexes to be defined on the table. In these cases, while the enhancement helps with logging resource consumption, it is not a great help in bringing down the response time of the delete. This is because the cost of updating these RID indexes dominates the total cost of the delete. This is analyzed and described in detail in [2]. Figure 5 which is taken from this paper shows the response time of a delete with various index clustering. Here the partkey and the receiptdate indexes have a cluster ratio of 4% and 38% respectively. While there is a 7 times performance improvement when we don't have a RID index defined, it drops to 33% with the receiptdate index and to 2% with the partkey index.

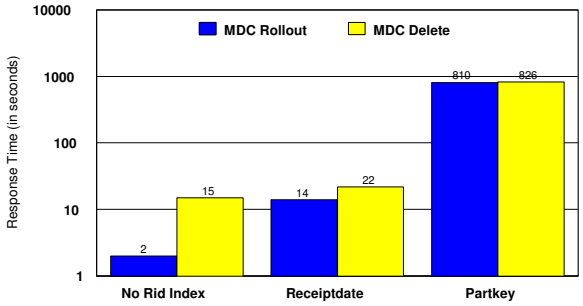

Figure 5: Response time of a MDC Rollout and Delete with different index clustering

The reasons for this can be deciphered from Figure 6 which is also taken from the same paper. It shows the logical and physical index page reads to be done as part of the index updates for rollout and delete. With the receiptdate index of 38% clustering, one ended up getting good bufferpool hit ratio for the index pages that were needed and thus the physical index page reads are lower. However, for the partkey index of 4% clustering, the amount of physical reads that needed to be done for almost the same number of logical reads was substantial. This accounted for the drop in response time for the partkey index.

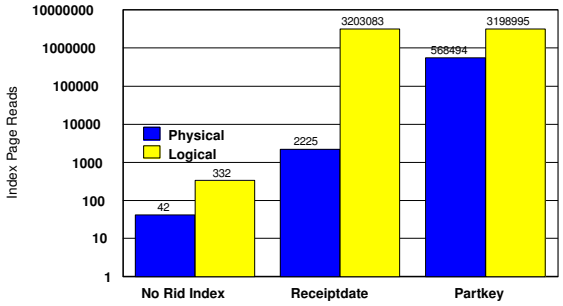

Figure 6: Index page reads for the delete in Figure 5

The problem of index updates will get really aggravated when we have multiple RID indexes defined on the table. To tackle this important issue, a major enhancement to MDC delete called Deferred Rollout was introduced in DB2 UDB Viper 2. Here we asynchronously update these RID indexes in the background while allowing the delete to commit and the table to be accessed. This results in a huge improvement to the response time of the deletes along with a major reduction of log footprint and physical IO on the indexes. This major enhancement is the focus of this paper.

4. THE CURRENT STATE OF THE ART

The delete mechanism employed by database engines generally works horizontally, on a row at a time. Each record is deleted and the defined indexes are updated one by one to reflect the delete of that record. For mass or multiple record deletes, one iterates over all records to be deleted in a similar fashion. The conventional MDC delete in DB2 UDB V8.1 is an example of that. While these are easy to implement, they are very inefficient for mass record deletes since the accompanying RID index updates results in random probes into the index. This translates to synchronous IO and is very costly.

Other technologies in this area include the Detach mechanism for range partitioned tables. Range partitioning is available in some commercial database systems like DB2 zOS and LUW [6], Oracle [7] and MS SQL Server 2005 [8]. In this, a table is partitioned into ranges of values on a specified attribute. Detaching a partition would be the equivalent of delinking all the data of the partition from the table. Any local indexes on that partition are also thrown out. If there are global indexes defined, these will have to be updated. Detach tends to be a Data Definition Language (DDL) level command and application have to explicitly specify they want to detach. This will, in most implementations, result in getting an exclusive lock on the table for the duration of the Detach. Thus, during the Detach, concurrent access to the table is generally disallowed.

Some database engines implement the base table in the form of a B+ tree itself [7] [9]. Here, additional secondary indexes are allowed and will have to be updated on a delete. In some implementations, to speed this up, multiple indexes could be updated in parallel [9] [10] [11]. There have also been recent works [12] on efficient online bulk deletion of the base B+ tree table itself. Here, all locks needed for the bulk delete operation are acquired during the scan of the leaf pages covering the target key range. However, the records qualifying the delete are marked for delete only. These records are then physically deleted in a later rebalance phase that avoids visiting subtrees in which all records qualify for deletion. It should be noted that this work did not focus on optimizing the update of secondary indexes which might be defined on the table.

A mechanism for bulk deletes was explained in [1]. The aim of this method was to improve the response time of the delete. This is an important consideration for mass deletes. However, it did not address the issues of resource consumption for logging or locking or the response time of the rollback of the delete. It also assumed the base table would be exclusively locked and the indices would be offline for the duration of the delete. The method described, is based on vertical deletes of the base table and any rid indexes defined on it. This is to be contrasted with the conventional method of deleting the table record and updating the rid indexes iteratively for all qualifying records.

Deferred maintenance was explored in [13]. Here, a differential file was used like a book errata list to identify and collect pending row changes. An up-to-date database view was effectively obtained by first consulting the differential file as the first step in data retrieval. In this scheme, one trades off increased access time for getting reduced database update costs. When the differential file grows sufficiently large, reorganization incorporates all changes into a new generation of the database.

It is to be noted that while not directly related to rollouts, there has been a lot of work on analysis and implementation of deletes on indices and related issues [14],[15]. Bulk load (also known as Rollin) is the opposite of Rollout. This has also been studied in a number of papers [16],[17],[18]. Deleting records from tables and the management of free space has been discussed in [19]

5. OVERVIEW OF DEFERRED ROLLOUT

The design goals of Deferred Rollout in DB2 UDB Viper 2 were to dramatically improve the response time of the Rollouts while keeping the table online and queryable. It also aimed to reduce the IO and logging involved in updating the RID indexes. The latter results in simplified applications since customers will now not have to break up their deletes into smaller parts using FFnRO (Fetch First n Rows Only) to work within their log space limitations.

Figure 7 shows a high level overview of our approach. When a delete that qualifies to be a rollout happens, the table records are deleted one by one as previously and one log record per page is written for them as before. However the RID indexes are not updated then and there with every record deleted. Rather, we mark the block they belong to as deleted in a new in-memory data structure called the Rollout Block Bitmap (ROBB). We also mark the corresponding entry in the on-disk block map with the InUse and Rollout bits turned on. When the cleanup of the records from the table is completed and the user wants to commit, the delete is committed as today. This obviously will leave the RID indexes as dirty with pointers in them to records which have been deleted. To prevent queries which use these RID indexes from returning wrong results, the ROBB is used to filter out accesses to these deleted records.

After the delete commits, Asynchronous Index Cleaners (AICs) are started in the background which go through these indexes and use the ROBB to identify deleted record entries, remove them from the indexes and log the index updates. This process will continue till it has cleaned all RID indexes of deleted record entries. Simultaneously the ROBB is also updated to mark the blocks cleaned as it happens. When cleaning is done, the ROBB is removed and queries using the RID indexes stop the extra step of filtering out records from them.

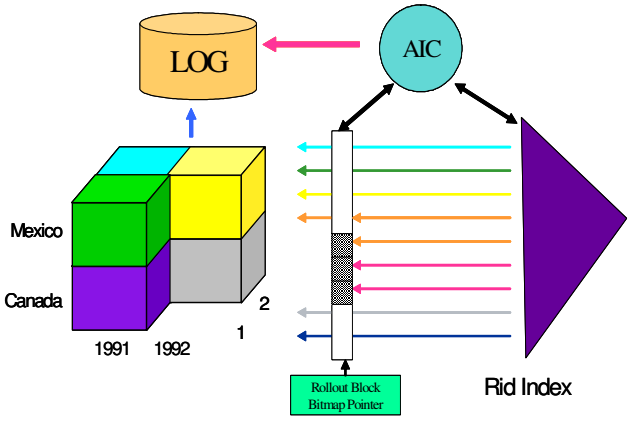

Figure 7: High level overview of Deferred Rollout

With this design, the tradeoff is that queries which use RID index scans will have to probe the ROBB to determine the state of the record while the indexes are being cleaned up. However table scans and block index scans will not be affected. Another tradeoff is that blocks will become available for reuse only when all the indexes have been cleaned up rather than as and when a block is cleaned up.

It is to be noted that this description of our approach is a very high level overview of the process and does not describe the concurrency and other issues that need to be addressed. In the subsequent subsections, we will describe each step and the challenges involved in greater detail.

6. ROLLOUT BLOCK BITMAP (ROBB)

The ROBB represents blocks which have been deleted from the table but have pointers in the RID indexes to them. A table can have up to 2^31 blocks and the ROBB needs to be able to handle them. As depicted in Figure 8, ROBBs are of two kinds namely the Master ROBB and the Local ROBB. The former represents the blocks which are deleted and committed but not yet cleaned by AIC. There is one Master ROBB at the maximum for a table object and it is used by all transactions to filter out access to deleted records from RID indexes in that table object.

The Local ROBB represents the blocks which have been deleted but not yet committed or rolled back. There is one Local ROBB for the table in every transaction which does a delete and is accessible only by it. It may represent the cumulative result of more than one delete in that unit of work. RID index accesses from this transaction will have to filter out RIDs not only from the Master ROBB but also from the Local ROBB. When the transaction commits, its Local ROBB is then merged into the Master ROBB for the table. This then becomes the single point of truth for all RID index based accesses from all transactions including this one.

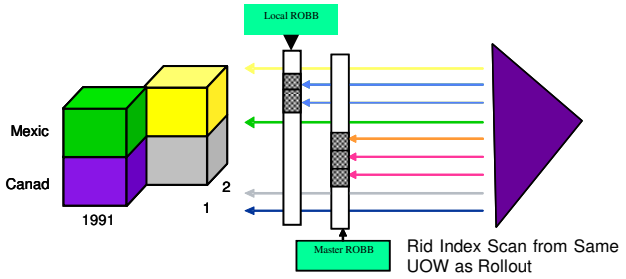

Figure 8: Table level Master ROBB and transaction level Local ROBB when there are committed and uncommitted Rollouts

6.1 Key Design Considerations

The key design considerations for the ROBB include:

1. Fast probes: Queries which use RID index accesses will be probing the ROBB on every RID to determine if the block the RID belongs to is deleted or not. A fast filtering mechanism is imperative to keep the query performance overhead low. This is especially true for index only scans where one accesses only the index and not the underlying table. This requirement means the ROBB has to be an in-memory structure and rules out a pure disk based structure.

2. Memory consumption: In real customer situations, one tends to have many tables and there could be multiple deletes going on at the same time. Thus the memory consumption of a ROBB is an

important design consideration. It has to be kept to the minimum possible. Since a table could have up to 2^31 blocks, a simple in-memory bitmap, where one has one bit for every possible block in the table pre allocated, is ruled out as impractical. Such a large bitmap would also mean that for random probes into the bitmap one would end up with a lot of data cache misses.

3. Commit/Rollout memory restrictions: Generally a commit or rollback operation happens without extra memory being allocated during that time. This is because an inability to get that memory could lead to a critical failure. Moreover, rollback is often done to free up resources and thus asking for more memory would go against the reason for doing the rollback. Keeping these in mind, ROBB operations done during these two periods will have to work within available memory.

6.2 ROBB Design

The ROBB design we choose is based on the principle that in most real life scenarios, only a part of the table will be deleted and thus the probability of the probes returning block deleted would be low. Given this, a hierarchical bitmap, where each level is tailored for a level in the memory hierarchy and the bottom level represents only deleted blocks was used for the ROBB. For example, the top most level (of size n bits) would fit a register. Registers are generally of sizes 32, 64 or 128 bits. Older machines had smaller registers and the newer ones have larger register sizes. The next level of the bitmap (of size m bits) would fit in the data cache of the machine. Machines could have multiple levels of the data cache like L1, L2, L3, etc., and we may have levels of the bitmap corresponding to one or some or all of the multiple levels of the data cache. The lowest level of the bitmap may fit in main memory.

FIG. 9 shows an example ROBB with four levels. Here the value of n and m is 64 and 8126 respectively. It is designed so that level 1 fits in a 64 bit register, level 2 in a data cache like L1 or L2, level 3 in a data cache of level L3 and level 4 would in main memory.

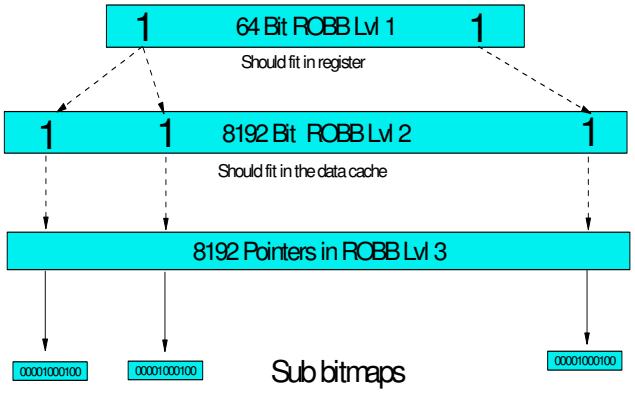

Figure 9: An example hierarchical ROBB

The lowest level sub bitmaps in FIG. 9 may have a maximum of m sub bitmaps each of round ((x/m) + 0.5) bits. A sub bitmap would physically exist only if one or more blocks it represented is deleted but not yet cleaned. In the example shown in Fig 8, there are three sub bitmaps materialized out of the m possible sub bitmaps. Each of them is shown to have 2 bits turned on. This means there are a total of 6 buckets deleted but not yet cleaned. The level above that (marked RobbLvl3) has m pointers to the sub bitmaps. They may be non NULL if the bitmap it points to exists. In the above example, there are three pointers which are not NULL. The levels above that (Robb Lvl2) have a bitmap of size m where there will be a 1 in a bit of position b if the pointer in the level below of that same number is non NULL. In this case there is three bits marked 1. The topmost level (marked RobbLvl1) has a bit s turned on if any of the bits ((s-1)* round (m/n) +0.5) to (s* round ((m/n) + 0.5)-1 of the level lower to it is on. In the example described in FIG. 9, there are two bits turned on in the top most level. The one on the left has two bits of the lower level corresponding to it turned on and one to the right has one.

Thus, in this representation, a bit in the top level represents a number of bits in the next lower level. If any one of the bits in the next lower level that the top level bit represents is turned on, then that top level bit is also turned on. The hierarchy may comprise any number of levels in similar manner depending on the design and system considerations.

The following are some of the operations which need to be supported on the ROBB:

6.2.1 Probe

When a ROBB probe via a RID index access happens, if the bit corresponding to the block at the top most level is off, it indicates the bucket is definitely not deleted and thus the query can proceed ahead. A scheme like this is ideal if the probability of a block being deleted is low, which happens in most real situations. It is to be noted that the level may likely be in a register and thus would be very fast to access if most of them return not deleted. If the register has a 1 for that bucket, then it would mean that there exists a probability of the bucket being deleted but not certain unless the lower levels are checked. In such a case, the lower levels are accessed iteratively until it hits the lowest level and gets a confirmation for the bucket or at some level it determines that the bucket is not deleted.

The probe is a very performance critical operation and happens very often. A RID index access will Probe the ROBB for every RID it encounters from the index.

6.2.2 Set and Clear

The Set and Clear operations on the ROBB are not very frequent. They will happen only when a block is deleted by a transaction or it is cleaned up by AIC respectively. So they are not very performance critical.

When a delete of a block happens, its bit in the ROBB will be set to 1. If the sub bitmap it belongs to at the lowest level does not exist, it will be materialized before the bit is marked 1. Further, the pointer at RobbLvl3 is set and the bits corresponding to the block at the higher level are iteratively turned on if they were not on already.

As the AIC cleans up the rid indexes and the blocks, it will clear the bit corresponding to the block in the ROBB. That will entail turning its bit at the sub bitmap in the lowest level to 0. If that completely empties the sub bitmap of bits marked 1, then the sub bitmap is freed and the pointer at RobbLvl3 is turned off. Further we iteratively move up the hierarchy and check if the bits at the higher level can be turned off. This can be done if all the bits corresponding to that bit in the lower level are turned off.

6.2.3 Merge and Subtract

A key design requirement for the ROBBs is that they should be easy to merge and subtract from each other. The merge operation is used at the time of commit when the transaction's Local ROBB is merged to the Master ROBB of the table. The subtract operation is used during the rollback of an in-doubt transaction when we need to erase the effects of a transaction from the Master ROBB. Since during commit and rollback, one cannot request for more memory, hence both the merge and subtract operations of the ROBB had to be designed to essentially work within the available memory of the Local and Master ROBB.

An "in-doubt" transaction is one whose fate (committed or rolled back) is not known during and immediately after a database restart. Its changes are tentatively reapplied to the database and locks are held on its behalf, until it is resolved. The reapplying of a Deferred Rollout results in the creation of a Temporary ROBB (recording all the blocks that were rolled out), which is retained until the transaction is resolved, and in the setting of bits in the Master ROBB. If the transaction commits, the Local ROBB is discarded. It if rolls back, the Local ROBB are subtracted from the Master ROBB. If the Master ROBB then becomes empty, it is discarded. Subtraction is simply unsetting all the bits in the Master ROBB that correspond to bits that are set in the Local ROBB. As sub bitmaps become empty, they are removed from the Master ROBB and corresponding bits at the higher level are unset.

To accomplish a merge, both the Local and Master ROBB come out of the same memory heap. First, if a Master ROBB does not already exist, the Local ROBB simply becomes the new Master ROBB. Otherwise, the Local ROBB is merged into the Master ROBB as follows:

Step 1: The new level 1 bitmap is replaced with the bitwise OR of the old master level 1 bitmap and the temporary level 1 bitmap.

Step 2: For each level that consists of a bitmap and corresponding pointers to lower level sub bitmaps, the bitmap is replaced with the bitwise OR of the old master bitmap at that level and the temporary bitmap at that level. For each pointer in the master bitmap's list of pointers, if the pointer is not null and the corresponding pointer in the temporary list is null, the pointer is not changed. If the pointer is null, the corresponding pointer is copied from the temporary bitmap to the master bitmap, and the lower level sub bitmap that it points to will belong to the master bitmap, from then on. If both the pointer in the master bitmap and the corresponding pointer in the temporary bitmap are not null, then the lower level sub bitmap is merged, either as in this step 2 (if it is also a combination of a bitmap and corresponding list of pointers) or as in step 3 (if it is the lowest level sub bitmap, consisting only of bits and no pointers).

Step 3: Each sub bitmap in the master for which a corresponding sub bitmap also exists in the temporary bitmap, as determined in step 2, is merged by bitwise ORing the two sub bitmaps, thus replacing the contents of the master sub bitmap with the result, and discarding the temporary sub bitmap (freeing the memory it occupies).

6.2.4 Recreate

This operation will recreate the ROBB during recovery after a system crash or normally when the AIC is suspended on a database deactivation. This is facilitated by the deleter marking every block it deletes with the Rollout bit in the on-disk block map and leaving the InUse bit on. When AIC frees up the block it will turn on the free bit of the block in the block map while resetting the InUse and rollout bits. At the time of a system crash, recovery will walk through the block map and will recreate the ROBB by setting the corresponding bit in it for every block it finds marked InUse and Rollout in the block map.

6.3 Prior Art on Hierarchical Bitmaps

Hierarchical bitmaps have been used in the past for Operating System memory management. Among other things they are used in some systems for allocation, deallocation, and reallocation of memory, and tracking the changes in the allocation states [20]. Hierarchical bitmaps have also been used in the communication industry ranging in applications from reporting reception result of packets [21], acknowledgement bitmaps in ARQ transmission protocols [22] as well as scheduling communication flows [23]. These types of bitmaps also find use in thread activity for multi processors as well as file system management [24]. Hierarchical Bitmap Indexes have also being used for indexing set value attributes [25].

It is to be noted that Hierarchical Bitmaps are a very generic term. For the sample applications mentioned above, a very specific flavor of Hierarchical Bitmap, tailor made for that application area was used in each case.

7. ASYNCHRONOUS INDEX CLEANUP (AIC)

The key aim of Asynchronous Index Cleanup (AIC) is reducing the physical IO involved in cleaning up RID indexes. As shown in Fig 6, in conventional delete, the physical IO could be very large for badly clustered RID indexes. Another aim is to reduce the amount of logging needed to cleanup the RID indexes. Customers go to the pains of dividing their deletes into smaller ones using FFnRO clauses to circumvent their log space limitations. Any mechanism which significantly reduces log space requirement will help simplify applications. Customers generally desire that these be done without taking the index or the table offline and while other rollouts on the table are allowed to happen. They also need to control the priority of the AIC or even stop it if necessary, to make way for high priority queries which might come in

To accomplish all these, when a delete commits and a Master ROBB is created, an AIC agent starts walking through the block map and stakes out its cleaning territory by marking blocks which have the InUse and Rollout turned on, with an additional cleaning bit. This allows it to differentiate between blocks it is cleaning and those which might be additionally rolled out while it is doing its job.

Subsequently one AIC agent is earmarked for every RID index defined on the table. This agent walks through the leaf pages of the index and for every RID it encounters; it probes the ROBB to determine if it is deleted. It removes deleted record entries from the index and for every leaf page it has changed, it writes one log record. After an extent has been thus processed, the AIC will commit its work. It will also prefetch index leaf pages ahead of where it is currently working to reduce IO waits. The agents will regularly also write its resume position into the index and log it. This is used to help a restart of the AIC. In this way, multiple

AIC agents work in parallel on the different indexes to clean them all up.

When the agents are done traversing their entire designated index, each agent waits at a wait-post till the last agent completes. The last agent out will walk through the block map and turn the blocks marked InUse, Rollout and Cleaning to free. It will also simultaneously clear the bit for that block in the Master ROBB. The blocks thus cleaned are now available for reuse. At this stage, the AIC is ready to do more work on the table. It will check the Master ROBB to see if any further rollouts have happened. If so, it starts all over again and takes care of all the pending rollouts in one single pass over the RID indexes.

While AIC is in progress, if there was a system crash, then recovery starts off with first rebuilding the ROBB using the Recreate operation described in section 6.2.4. Then the AIC agents are started off cleaning from the last resume position written in the index. This way the previous work is preserved.

This design fulfills all the initial aims of AIC. It trades off random probes of the index with a traversal of the index leaf pages and semi ordered probes of the ROBB. Except for very tiny deletes, this scheme will win in physical IO over the conventional scheme. Even in that case, if multiple tiny deletes can be combined during the cleanup of the indexes by AIC, it will likely win. It also converts a per-RID logging of the index update to a per index page logging. Further, it allows the indexes to be processed in parallel. In DB2 LUW, apart from MDC tables, AIC has also been applied to Range Partitioned tables. Range Partitioned tables can be used in conjunction with MDC.

This AIC design needs to be contrasted with just doing the job of index updates asynchronously in the background using the conventional "row at a time" mechanism. This could be done by walking through the block map and farming out full blocks to AIC agents to cleanup. For every record in the block, the agents could update the RID indexes iteratively. While this could help with reducing the response time of the rollout, it will not help in lowering the IO involved in updating the indexes nor would it reduce the log space.

8. EXPERIMENTAL EVALUATION

For the experimental evaluation, we used a setup similar to that used by some customers who run ERP solutions over MDC tables. We used a 2 dimensional MDC table with 9 RID indexes and 3 system defined block indexes on them. One of the RID indexes was a unique index. The table and indexes were defined on different tablespaces but shared the same bufferpool. This setting is common to some customers running ERP on MDC tables. Table 1 provides more details of the experimental setup.

The evaluation was done using deletes with predicates on the time based MDC dimensions. Time is often used as an attribute for bulk deletes. The deletes ranged in size from 0.3% to 97% of the table. In this evaluation, all figures marked "Delete" in the charts denote the conventional "record at a time" delete. The Immediate Rollout denotes the algorithm used in DB2 V8.2.2 Saturn which was described in Section 3. The algorithm which is the focus of this paper is marked as Deferred Rollout. The figures marked Deferred Rollout + AIC is the cumulative time for the delete and for AIC to do the cleanup.

To evaluate the design, a study of the response time of the rollout and the IO involved was clearly important. Equally important for an online mass delete mechanism are parameters like amount of logging. Also of interest is how queries which use RID index scans will behave when the indexes are dirty after the rollout and need the ROBB filtering. It is to be noted that locking is not being evaluated here since there is no change in locking between Immediate Rollout and Deferred Rollout. All MDC bulk deletes mechanisms tend to take a fraction of the locks that a non MDC delete would require [2].

Table 1: Experimental setup details

Hardware System	IBM 7028-6C4 with 16GB of main memory
Processors	4 x 64 bit PowerPC_POWER4 @ 1453 MHz
L1 Data Cache	32KB
L2 and L3 Cache	1.44MB and 32MB
Operating System	64 bit AIX 5.3.0.0
DB2 Instance	DB2 UDB Viper 2 with 4 MLNs
DB2 Registry Variables	DB2_MDC_ROLLOUT=DEFER/YES/NO
Tablespace Details	Page size of 16KB; Extent size of 16 pages
Table size	11 million rows in 134260 pages
Index sizes	Unique index (just 1 RID per index key) : 32716 pages Non unique indexes : ~ 4700 pages each
Index Cluster Ratios (degree to which table data is clustered in relation to this index)	3 RID indexes with below 5% clustering 2 with clustering in the range of ~35% 4 with above 95% clustering.

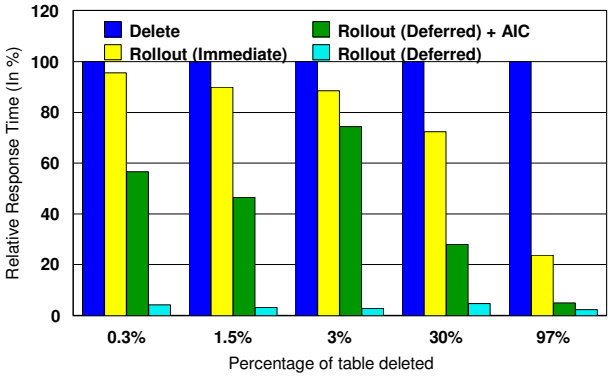

Figure 10: Response time of the rollouts

As seen in Figure 10, the response times of the deletes improve about 25 times with Deferred Rollout over Deletes. Even in comparison to Immediate Rollout, it is at least an order of magnitude faster. This is due to the costly index updates being done later. If one were to include the cost of the index updates done in the background (as seen in Deferred Rollout + AIC), one still sees huge response time improvements ranging from 2 to 5

times. The gains are very heavy for the large rollouts and tend to lessen for the smaller once.

The improvements are partly due to the fact that with AIC, we are very successful in prefetching the RID index pages for the index updates. This is not done in the conventional "record at a time" updates mechanism employed by both Delete and Immediate Rollout. Prefetching the index pages results in the 2 to 25 times lower IO waits seen in Figure 11. The IO wait decrease is more pronounced for the medium to large deletes. With pages now more easily available, the agents can do more useful work and that has a positive influence on the response time.

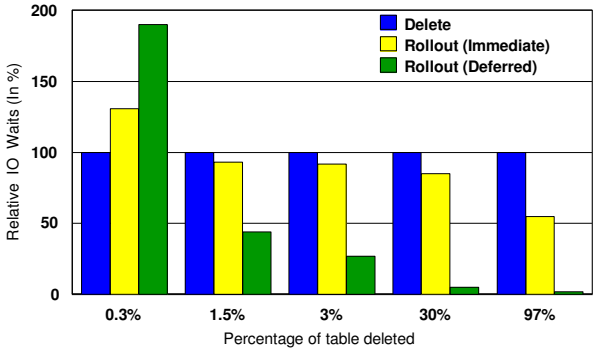

Figure 11: IO Wait profile during rollouts

We also replace the random probes into the index with a sequential scan of the index pages. This means the number of times an index page is needed is dramatically reduced. This is visible in Figure 12 which shows the index logical reads. The lower requirement for the page will mean lower load on the system and better performance.

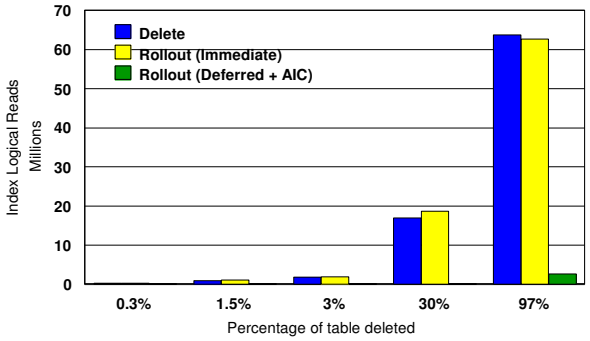

Figure 12: Index Logical Reads of the rollouts

System performance is also influenced by the reduction in the amount of logging we have to do. Figure 13 shows the log space consumed by the deletes. For most of the cases, the log space consumed is 3 to 20 times lower for the Deferred Rollout in comparison to the Rollout Immediate. The figures for the standard Delete are much higher. These gains are due to writing one log record for every index page updated rather than one for every index rid being updated.

Lower log space consumed will translate to lower number of transactions which might rollback due to log full. It will also mean applications will not have to use FFnRO to break big deletes into smaller parts to work with available log space. This will make writing applications simpler apart from having a positive impact to the overall system performance.

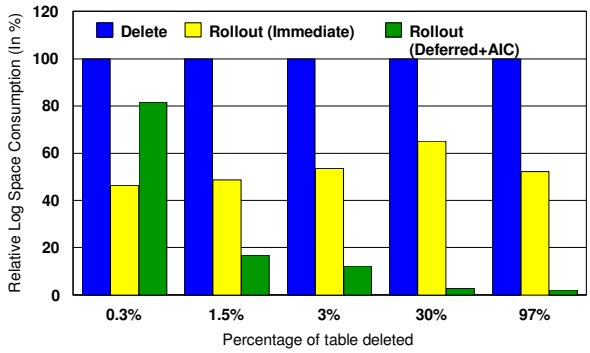

Figure 13: Log space consumption of the rollouts

To analyze the impact of Deferred Rollout on query processing, a set of delete followed by query sequences were executed for both Deferred Rollout as well as Immediate Rollout. The queries were exercising table scans, block index scans, RID index scans as well as RID index only plans. The timings taken were for both the deletes and as well as the queries. In the case of Deferred Rollout, the queries and the AIC cleanup overlapped to some extent. This comparison would represent what a user would experience while using both methods. As we see in Figure 14, the queries finished with significant gains for the Deferred Rollout case in all query plans. For the RID index only query, the timing was 80% better. For the RID index scan queries, the numbers were about 15% better. Both these types would probe the ROBB. The CPU overheads of the ROBB probes turned out to be a tiny fraction of the total CPU cost of the RID index scan queries. For badly clustered indexes it was in the order of 3% and for others it was in the order of 1-2% only.

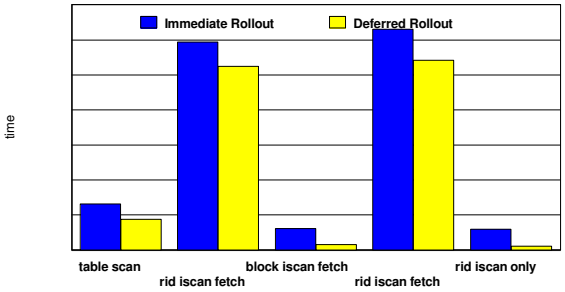

Figure 14: Query performance after deletes with both timed

9. CONCLUSION

The MDC Deferred Rollout mechanism provides a very efficient and usable online bulk delete mechanism for MDC tables in DB2 Viper 2. Apart from providing a fast response time for deletes, it also helps keep the logging and locking resource consumption low. This enhances concurrent read/write access to the table by dramatically reducing the chances of lock escalation and out of log space situations while the delete is in progress. It also enables simplification of applications by not needing big deletes to be broken into smaller parts using mechanisms like FFnRO etc and by being able to work out of the standard DML delete statements.

In this paper, we have detailed the design of MDC Rollout and the challenges that needed to be addressed. We also shared performance results which show that MDC Deferred Rollout meets its design goals very efficiently.

10. REFERENCES

[1] Gartner, A., Kemper, A., Kossman, D., Zeller, B., "Efficient Bulk Deletes in Relational Databases", Proceedings of the ICDE 2001

[2] Bhattacharjee, B., "Performance Study of Rollout for Multi Dimensional Tables in DB2", First ACM International Workshop on Performance and Evaluation of Data Management Systems (EXPDB) 2006

[3] Padmanabhan, S., Bhattacharjee, B., Malkemus, T., Cranston L., Huras, M., "Multi-Dimensional Clustering: A New Data Layout Scheme in DB2", Proceedings of SIGMOD 2003.

[4] Bhattacharjee, B., Padmanabhan, S., Malkemus, T., Lai, T., Cranston, L., Huras, M., "Efficient Query Processing for Multi-Dimensionally Clustering Tables in DB2", Proceedings of VLDB 2003

[5] Lightstone, S., Bhattacharjee, B., "Automating the design of multi-dimensional clustering tables in relational databases", Proceedings of VLDB 2004

[6] http://www-306.ibm.com/software/data/db2

[7] http://www.oracle.com

[8] http://www.microsoft.com/sql/default.mspx

[9] http://www.hp.com/go/nonstop

[10] Leslie, H., Jain, R., Birdsall., D., Yaghmai, H., "Efficient Search of Multi-Dimensional B-Trees", Proceedings of the VLDB 1995

[11] Englert, S., Gray, J., Kocher, T., Shah, P., "A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases", Technical Report 89.4, Tandem Part No 27469, May 1989

[12] Lilja, T., Saikkonen, R., Sippu, S., Soisalon-Soininen, E., "Online Bulk Deletion", Proceedings of ICDE 2007

[13] Severance, D.G, Lohman, G.M, "Differential Files: Their Application to the Maintenance of Large Databases", ACM Transaction on Database Systems, Vol. 1, No. 3, 1976

[14] Jannink, J., "Implementing deletion in B+trees", SIGMOD Record, Mar. 1995.

[15] Johnson, T., Shasha, D., "B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half", Journal of Computer and Systems Sciences, 1993

[16] Van den Bercken, J., Seeger, B., Widmayer, P., " A generic approach to bulk loading multidimensional index structures.", Proceedings of the VLDB 1997

[17] Wiener, J., Naughton. J., "Bulk loading into an OODB: A performance study", Proceedings of the VLDB 1994

[18] Wiener, J., Naughton. J., "OODB bulk loading revisited: The partitioned-list approach", Proceedings of the VLDB 1995

[19] McAuliffe, M., Carey, M., Solomon, M., "Towards effective and efficient free space management", Proceedings of the SIGMOD, 1988

[20] Forin et al, "Hierarchical bitmap-based memory manager", Patent no. 6175900, Microsoft Corp

[21] Jang et al, "Method for reporting reception result of packets in mobile communication systems", Patent application 20060034277, Samsung Electronics Co Ltd

[22] Chen et al, "System and methods for implementing hierarchical acknowledgement bitmaps in an ARQ protocol", Patent no. 6658619, Ericsson Inc

[23] Ward et al, "Mechanism for efficient scheduling of communication flows", Patent no. 6037611, Sun Microsystems Inc

[24] Burgess et al, "Method and system for monitoring file attributes using bitmaps to determine group membership of files and to determine which files have been processed", patent no 5504889, Microsoft Corp

[25] Morzy, M., Morzy T., Nanopoulos, A., Manolopoulos, Y.,"Hierarchical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attributes", 7th East European Conference on Advances in Database and Information Systems, 2003