Recovery Principles of MySQL Cluster 5.1

Mikael Ronström Jonas Oreland

MySQL AB
Bangårdsgatan 8
753 20 Uppsala
Sweden
{mikael, jonas}@mysql.com

Abstract

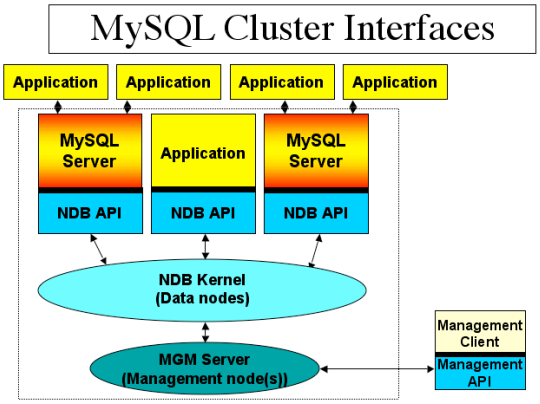
MySQL Cluster is a parallel main memory database. It is using the normal MySQL software with a new storage engine NDB Cluster. MySQL Cluster 5.1 has been adapted to also handle fields on disk. In this work a number of recovery principles of MySQL Cluster had to be adapted to handle very large data sizes. The article presents an efficient algorithm for synchronizing a starting node with very large data sets. It provides reasons for the unorthodox choice of a no-steal algorithm in the buffer manager. It also presents the algorithm to change the data.

1. Introduction

MySQL Cluster is a parallel main-memory DBMS. In MySQL Cluster 5.1 fields on disk are introduced. This introduces a number of challenges on the recovery architecture.

1.1 Description of MySQL Cluster

MySQL Cluster uses the normal MySQL Server technology paired with a new storage engine NDB Cluster. Data within MySQL Cluster is synchronously replicated among the data nodes in the cluster. MySQL Cluster uses the shared-nothing architecture, data nodes in the cluster handle their own storage and the only means of communication between the nodes is through messages. The main reason for choosing a shared-nothing architecture is to enable a very fast fail-over at node failures. It also doesn't rely on an advanced storage system as most shared-disk systems do. This means that normal cluster hardware can be used also for building large database clusters.



Internally there are a number of protocols designed, to be able to handle single failures most of these protocols have an extra protocol to handle failures of the master and to be able to handle multiple failures there is also an extra protocol to handle failure of the new master taking over for the old master.

Applications can use the normal MySQL interface from PHP, Perl, Java, C++, C and so forth, the only difference is that the tables are created with ENGINE=NDBCLUSTER.

In version 4.1 and 5.0 of MySQL Cluster all data in MySQL Cluster resides in main memory distributed over the data nodes in the cluster.

1.2 Challenges for Node Restart

Main memory is fairly limited in size. A starting node is synchronised by copying the entire data set to it. Even a large node with 10 Gbyte of data can in this manner be synchronised in 10-20 minutes. Using fields on disk the size of the data set in a data node in MySQL Cluster can easily be 1 TB. The current synchronisation method will thus take one day. This is obviously too much. A solution

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

is needed that restores an old copy in the starting node and only synchronises the changes since the node failure.

1.3 Buffer Manager

A main memory database has no buffer manager at all since data resides in memory. Since the size of a main memory database is small, it is possible to write it to disk in its entirety when performing a checkpoint. This enables a lot of optimisation on the logging.

As in any modern DBMS, the no-force algorithm [1], [6] is used. This means that at commit it isn't necessary to write pages to disk.

A traditional DBMS also uses the steal algorithm [1], [6], with in-place updates. Whenever a page needs to be written to disk, it is written and the proper log records are produced to ensure that any uncommitted data on the page can be undone.

In our first step towards disk data in MySQL Cluster it was natural to keep indexes in main memory and add the option that certain fields are stored on disk. Thus each record has a main memory part and a disk part.

In a traditional DBMS every update will be written to disk at least three times with each write containing a certain amount of overhead. The first write goes to the REDO log to ensure the committed data is not lost, the second write goes to the UNDO log to ensure we can restore a certain point in time from where the REDO log can be applied (many variations on this theme exists in DBMS's). The third write is to update the page where data resides whenever it is flushed from the page cache (this write to disk can disappear since data is cached if several writes of the same data occurs before the write goes to disk).

Our design goal was to find an algorithm that can perform updates with only two disk writes. MySQL Cluster is designed for modern cluster computers with large main memories. Thus the no-steal algorithm is of great interest. With the no-steal algorithm there is no uncommitted data written to disk and thus there is no need to undo changes in recovery. Thus the write to the UNDO log can be avoided. The no-steal has the disadvantage of using more memory space for the buffer manager. There are disadvantages particularly for large transactions. Our design is optimised for the transactions that are fairly limited in size although still striving for correctness for a large transaction.

1.4 Related Work

A rather old article from Computer Corporations of America [7] discusses much of the basics of recovery in shared-nothing architectures. In [8] Hvasshovd explain some new ideas on how to perform recovery of a distributed DBMS. In [2] Ronström explains an alternative approach to recovery of a distributed DBMS when it is known that all data resides in main memory. This work is an extension of that work where some parts of the data are moved to disk.

2. Recovery Principles in MySQL Cluster

To enable fast restart of the cluster, MySQL Cluster has a requirement that REDO log records can be used from other nodes in the cluster. This means log records cannot be connected to pages in the local node. The log records must be operational log records, meaning they specify only the operation and what was changed (e.g. operation: UPDATE, action: set A=2). As all DBMS's with logging it is necessary to regularly checkpoint the data pages to disk. Due to the requirement on operational log record MySQL Cluster checkpoints (called Local Checkpoint, LCP) must restore a database where all references between pages must be correct.

MySQL Cluster was designed with telecom applications in focus. These applications have tough requirements on response times, also on updates. At the same time they have very high requirements on reliability and availability. To meet those requirements the commit protocol was divided in two phases. The first phase commits the data in main memory of all nodes where data resides. The second phase collects a number of transactions together and ensures that all of them are synched to disk and restorable in case of a complete cluster crash. The second phase is called Global Checkpoint, GCP.

3. Node Restart Algorithm

The original algorithm in MySQL Cluster does a complete resynch and copies all of the data from the live node to the starting node. During the resynch all updates are performed on all copies to ensure that data is up-to-date when copy process is completed. For data that resides on disk it is not a feasible solution to copy all data.

The first idea that pops up is to restore the starting node from its own disk to the point where it crashed. Then the problem is how to synchronise the live node and the starting node. This synchronisation problem is the main issue discussed in this section.

3.1 Initial idea

The original design idea was based on an analysis of the possible differences between the live node and the starting node. This analysis goes through UPDATE, DELETE and INSERT respectively.

UPDATE provides no other problem than a TIMESTAMP problem to ensure that it is possible to avoid synchronising those records that haven't changed (normally the majority of the records).

INSERT means that the live node consists of new records not existing in the starting node. This is also easy since they have a new TIMESTAMP and can be handled by a scan of the live node.

DELETE means that the starting node can contain records no longer present in the live node. This is the major difficulty of the synchronisation problem. This cannot be solved through a scan of the live node. It can be solved through a scan of the starting node. This requires sending a verification of each record in the starting node to verify its presence in the live node. This is very time consuming and almost as bad as restoring from scratch.

The next idea that pops up is to introduce a DELETE log in the live nodes. Thus before synchronising the nodes, the DELETE log is executed from the GCP that the starting node has restored. After this execution the starting node contains no records not also present in the live node and thus a scan of the live node with synch of updated/inserted records will be enough to bring the starting node in synch with the live node.

This method works but has the unsatisfactory side effect that a new DELETE log is introduced. This has negative effects both on performance and on design complexity. Thus new solutions were looked for.

3.2 iSCSI idea

A new idea of how to solve the resynch problem came when considering the same problem for another application.

The problem here is a cluster of computers implementing an iSCSI server. iSCSI is an IP protocol for accessing disk subsystems. It implements a SCSI protcol over TCP/IP [4].

SCSI is much simpler than SQL. The only methods needed are READ, WRITE and defining and undefining logical disks. We avoid all other details of the SCSI protocol here.

SCSI is a block-based protocol, so it is below the level of files. Thus in our model it consists of records of fixed size equal to the block size and a table can be mapped to a logical disk. On these blocks on the logical/physical disks it is possible implement databases, file systems and so forth.

Now a resynch here is much easier. The first step is to get hold of the logical disks present in the live node. The number of such logical disks is limited and is thus not a very time-consuming activity. So for this part one can use the full copy as in the current algorithm.

The second step is to resynch the blocks. Since only WRITE's change data and the WRITE is similar to an UPDATE it is enough to check the blocks on a timestamp to perform the synchronisation. Thus a scan of all blocks in the live node and synch of those updated is enough to bring the starting node in synch with the live node.

3.3 Final Algorithm

The same idea can be mapped also to an SQL database. Instead of defining primary keys as the record identifier we use ROWID's as record identifer. The ROWID consists of a table identity, partition identity, page identity and a page index. In many cases the table identity and partition identity is known and is not needed.

Mapping the SCSI algorithm to an SQL database thus becomes:

1) Synchronise one partition at a time (done already in current MySQL Cluster version)
2) Specify number of pages in the partition in the live node
3) Delete any tuples in the starting node that is residing in pages no longer present in the live node.
4) For each possible ROWID in the live node check its timestamp and synchronise with the starting node if needed.

After these steps the starting node is synchronised with the live node. Step 4) requires some further small tidbits to work and to be efficient. At first it is necessary to resynch ranges of ROWID's in cases where a range of ROWID's are not used anymore (as in the case of a page no longer used in the page range between start and end page identity). Second it is necessary to keep a timestamp also on ROWID's deleted. When this ROWID is reused it will obviously also update its TIMESTAMP but it is important also that the TIMESTAMP exists also before the ROWID is reused. If the ROWID is reused for an internal record part, the old timestamp must be retained.

The following logic will be used in the resynch process. This logic executes on the live node that is to resynch its state with the starting node. Before starting the resynch logic, the live node need to know the TIMESTAMP that survived the crash in the starting node. The copy process will proceed page by page until all pages have been processed.

For each page one must keep track of the maximum ROWID used in the page. This number must never decrease unless all copies decrease them in a synchronised manner. For the presentation we assume it never decrease.

1) If there are no ROWID entries that have a newer TIMESTAMP, the page hasn't changed and no synchronisation is needed. In this case also the maximum ROWID of this page on the starting node must be the same since row entries have changed.

2) If most rows are up-to-date then only the records that have changed are sent to the starting node. In this case full records are sent through INSERT row entry unless the row entry indicated that the row is deleted in which case a DELETE row_id is sent to the starting node.

3) In some cases all entries in a page have been removed since the starting node crashed. In this case it is possible to send a DELETE RANGE start_row_id end_row_id. It is not possible to use ranges that span over pages since it is not possible for the starting node to know what the maximum ROWID in a page is. This could have changed since last. However it is possible send multiple ranges in one request.

4) Many times new pages are allocated in chunks. In this case there might be a set of empty pages in the live node that doesn't exist in the starting node. These pages can be synchronised by sending a specific EMPTY PAGE message with the page identifier.

New pages in the live node filled with new records will fit in case 2) since most or all entries are changed since the starting node crashed. In cases where massive deletes have occurred a combination of case 2) and 3) is used and finally massive updates.

3.4 Scan Order in Live Node

One performance problem to consider for this new algorithm is how to scan in the live node. There are two methods. The first is the simple method to simply scan in ROWID order. ROWID's are referring to main memory pages so the scan is performed as random access from the disk data's point of view. Thus each record that needs to be copied over follows its reference to the disk data part and reads that part to send it.

In situations where most data has been updated since the starting node crashed it is more optimal to scan in disk order instead of in ROWID order. In this case the pages on disk are scanned and for each record found on disk the main memory part is checked to see whether it needs to be sent or not. After scanning the disk it is also necessary to make another scan in ROWID order to find deleted records and new pages with still unused row entries. For this algorithm to work there has to be a bit set in the record header that indicates whether a record has been sent already or not in the scan in disk data order.

3.5 Recovery during Node Recovery

During the recovery process the starting node has to be careful so as not to destroy the capability to restart node recovery if the recovery fails. For this to be possible it is necessary to perform UNDO logging but REDO logging is not needed. After completing the node recovery process REDO logging is restarted and the node participates in an LCP to ensure that the node can be recovered to the new point in time.

3.6 ROWID uniqueness

The use of ROWID as record identifier introduces a new requirement to ensure that all replicas use the same ROWID. This is accomplished by ensuring that the primary replica always picks the ROWID and since MySQL Cluster updates the replica one at a time this is simple. The only problem is in certain situations when the primary replica is changed. Then it is necessary to be careful to avoid using the same ROWID for two different records.

3.7 Analysis of Algorithm

A traditional algorithm to perform synchronisation is by replaying the REDO log of the live node on the starting node. If the crash was short and not many updates have occurred then this algorithm is very fast.

The problem of this algorithm is

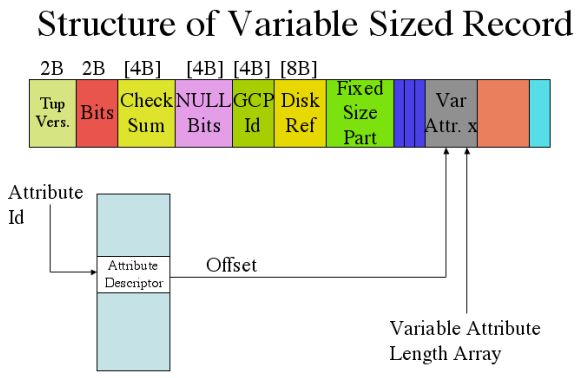
- Difficult to synchronise after a longer period of unavailability of starting node.
- Very difficult to avoid a write lock of system at the end when log replay has been performed and need to go over in normal operational mode. Problem is similar to jumping on a train that moves, it is easy to come up alongside of the train but very difficult to embark the train while it is moving.
- If many updates are made to the same record we have to reapply all updates, not just the last since the log has no an index of its history and future.

So our analysis shows that the new algorithm has a very stable performance independent of the time between the crash and the recovery. It doesn't require any massive locks that decrease concurrency of the system.

The scan of only main-memory parts should be more efficient as long as no more than 5% of the disk data has changed since the crash. In the case where more data has changed then a scan of the disk data is superior compared to the random access caused by scanning the main memory parts.

4. Record Layout

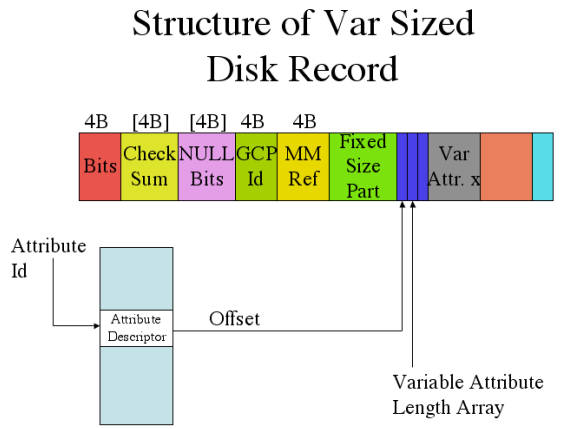
In the figure below the layout of the main memory part of a record with variable sized fields are shown. The ideas on the record layout contains no new ideas, most of the possible structures are shown in [5]. It is shown here to improve the understanding of the rest of the paper.



The checksum is a field used to store a checksum over the record in main memory. It is a field that is prepared to be an optional field that currently is mandatory. The

NULL bits contain NULL bits of all fields in the main memory part (if there are any NULLable fields). The GCP id is the TIMESTAMP used by the node recovery algorithm. It is not needed for main memory tables that can handle copying all records in the table at node restart. The disk reference is a page identity and page index in the disk part. Most parts of the record header will be optimised for storage at some point in time.

The disk part has more or less the same fields although the tuple version and disk reference is not useful. There is instead a reference back to the main memory part.



In case a completely minimal main memory part is desired, it is possible to decrease the header part to only contain the GCP identity plus a minimal header. Without this TIMESTAMP the node restart will be not be fast enough as already shown. Tuple Version can be removed if it is known that no indexes are used on the table. The bits header can be very small and indicate what parts that are present. For fixed size main memory part and fixed size disk part it is possible to use a special mapping from ROWID to disk ROWID to remove the need of the disk reference. NULL bits aren't needed if there are no NULLable fields.

To enable retaining the GCP identity (TIMESTAMP) for deleted records the GCP id is kept in the directory of the variable sized pages where a reference to the record and its length is previously stored.

5. Checkpoint Algorithm for Main Memory

As explained previously MySQL Cluster uses a REDO log to replay transactions not yet written to disk. To ensure that the REDO log is not of infinite size LCP's are performed. When going through these mechanisms for disk fields we also optimised the behaviour of main memory fields. Rather than saving the memory pages in the LCP we opted for saving the records instead. This is an acceptable method since all indexes are rebuilt at recovery.

The algorithm for checkpoints of main memory data is thus very simple. A full table scan is performed reading the latest committed version of each record. The same reasoning as shown in [3] proves that applying a REDO log onto this checkpoint will restore a consistent version of the table as long as the REDO log is applied from a point that is certain to be before the time of the start of the LCP.

Now this simple algorithm is enough for pure main memory tables. For tables with mixed main memory and disk content an extra requirement exist. The disk data is checkpointed using a traditional page approach and it is restored to the point where the checkpoint was started. It is important that the LCP of main memory data is restored to the same point in time. The reason for this requirement is that there is a reference from the main memory record to the disk part. The recovery system must ensure that this reference is valid also after recovery.

To accomplish this a simple idea is used. It is a traditional copy-on-write algorithm using a single bit in the record header. The idea is to get a consistent checkpoint from the point in time of the LCP start. If anyone tries to modify the record before it has been scanned, the record is copied to storage ensuring that the record is saved in the state at the time of the LCP start. This is a popular method to achieve a consistent state. One bit per record is needed to keep track of whether the record have been scanned yet. Whenever a commit of an UPDATE/DELETE is performed the following check must be made:

If the bit is set to 0 and an LCP is ongoing and this record is not yet written, write the record to the LCP storage at commit of the update and set the bit to 1. If the bit is already set to 1, no action is performed.

At Commit of an Insert the bit is set to 1 if an LCP is ongoing and the record has not been scanned yet.

When the LCP scan passes a record it checks the bit. If it is set to 1 it ignores the record and sets the bit to 0. If the bit is set to 0 it reads the record as part of the scan.

The algorithm requires that the state of the LCP scan has a defined point and that it is possible to check if a record has been scanned yet. In this implementation it is accomplished through scan in order of the ROWID.

As with all implementations that write to storage through a buffer, there can be problems when the buffer is overrun and this is yet another such possibility. The same problem previously occurred also with writing of the UNDO log. To solve this the commit is delayed until there is buffer space available.

To clarify the concept three examples are used. The following events happen (in time-order) for record r1.

Example 1: Update r1, Start LCP, LCP scan r1, Update r1, LCP Completed.

Example 2: Start LCP, Delete r1, LCP scan r1, Insert r1, LCP completed.

Example 3: Start LCP, Insert r1, LCP scan r1, Update r1, LCP Completed.

It is enough to exemplify using one record since each record is handled independent of all other records.

In the first example the first update will find the bit set to 0 since this is always the case when no LCP is ongoing, and since no LCP is ongoing no action is performed. When the LCP scan reaches the record nothing has occurred and so the bit is still 0, thus the record is read as part of the LCP and the bit is not changed. Finally when the last update is done the bit is set to 0 and LCP is ongoing but the record is already written. This is discovered by keeping track of whether LCP ROWID > r1 ROWID.

In the second example the record is deleted after starting the LCP. In this case the record must be written to the LCP before deleting it. It is also required to set the bit in the record header even though the record is deleted. The LCP scan will detect either that the record is deleted or that it changed and was processed previously, in both cases no action is needed by the LCP scan. Later the record is inserted again (it might be a different record but using the same ROWID) and since the LCP scan has already passed the record the bit is set to 0 when inserting the record.

Finally in the third example we insert a record before the record has been scanned. In this case it is necessary to set the bit to 1 when inserting the record but not writing it to the LCP since the record was not present at start of LCP. When the record is reached by the LCP scan it will find the bit set and will reset it with no record written to the LCP. When update is performed after LCP scan it will detect the bit set to 0 and will detect that LCP scan already passed and will ignore both writing the LCP and the bit.

6. Index Recovery

In the original design of MySQL Cluster the hash index was restored at recovery of the cluster. This design has been changed so that the hash index is rebuilt at recovery. With the new algorithm as described this will happen automatically during restore of a LCP. The LCP is restored through insertion of all records.

Ordered indexes are restored at recovery. They are not built automatically while inserting the data. To enable best possible performance of the index rebuild this is performed one index at a time. The most costly part of index rebuild is cache misses while inserting into the tree. This is minimised by building one index at a time.

At the moment all indexes in MySQL Cluster are main memory so rebuilding indexes at recovery is ok for the most part. If it is deemed to be too much work, it is very straightforward to apply normal recovery techniques of it. Especially for the unique hash indexes, as they are normal tables and no special handling at all is needed for recovery of those.

7. Checkpoint Algorithm for Disk Data

Fields on disk are stored on disk pages in a traditional manner using a page cache. Allocation of pages to a part of a table is done in extents and each part of the table grabs an extent from a tablespace defined at table creation.

Recovery of disk data is relying on the REDO log in the same manner as the main memory data. The disk data must recover its data from the point in time at the start of the LCP. In particular all references between pages in disk part and between main memory part and disk part must be exactly correct as they were at the time of the LCP start.

In order to produce a checkpoint that recovers exactly what was present in the disk part at LCP start we are using the same method as is used currently in MySQL Cluster. We checkpoint all pages (for disk data only changed pages) after LCP start and UNDO log records are generated thereafter to enable us to restore the data as it was at LCP start. For disk data UNDO logging must proceed even after completion of LCP since all normal page writes are changing the pages on disk. The UNDO log records are physiological log records [5] meaning that they specify the logical changes to a page (thus reorganisation of the page need not be UNDO logged).

Before writing a page from the page cache to disk it is necessary to conform to the WAL (Write Ahead Log) principle [5] meaning that all UNDO log records must be forced to disk before the write can proceed.

Now an observation is that it isn't really necessary to UNDO log all data changes. Whatever data changes was applied after LCP start will also be present in the REDO log and when executing the REDO log the contents of the record will never be checked, the records are simply overwritten with their new content. Thus UNDO log records need only report changes of structure (insert new record parts, delete record parts and move of record parts).

There is a complication however. The problem is that our algorithm writes to the pages at commit time. At commit time however the data is not yet synchronised with disk, so only at completion of the GCP of the commit is it ok to write the page to disk. Now this complication of the page cache algorithm is unwanted so an alternative method is to filter the UNDO log before writing it to disk.

The solution to this problem is to write full UNDO logs as usual to the UNDO log buffer. At a time when it is decided to send UNDO logs to disk each log record is checked to see,if the data part in it is part of a transaction whose GCP is already synched. If it is, it will be skipped. If it hasn't been synched yet, also the data part is written to the UNDO log. In this manner complex logic around page writes is avoided. Still, for the most part, the data part is removed from the UNDO log.

One major reason of not complicating the page cache logic is the large transactions. These transactions will at

commit start writing massive amounts of pages and if it is not allowed to write those to disk very soon the page cache will be swamped and the system will get into a grinding halt. So again large transactions will cost more than small transactions, but they will be doable.

8. Abort handling Decoupled from Recovery

One distinct advantage that we have accomplished with the recovery algorithm is that a full decoupling of recovery logic and abort logic is achieved. The algorithm for UPDATE and INSERT are shown to exemplify how this is achieved.

8.1 UPDATE algorithm

When an UPDATE is started the main memory part is found and through that a reference to the disk part as well. Before the UPDATE starts it is ensured that all disk pages used by the record is loaded into memory. An optimisation is that the disk page is not loaded if it is known that the UPDATE only will touch main memory fields.

The UPDATE is not performed in-place, rather a new region of memory certain to contain the record is allocated. This is allocated from "volatile" memory not surviving a crash and not part of any recovery algorithms. The current data in the record is copied into this "volatile" memory. Next the UPDATE is executed. This could involve using interpreter code internal to the data node and can involve any fields.

When the UPDATE is completed, the size of the variable parts in main memory and disk are calculated. The space on the page currently used is pre-allocated if possible. If not possible, space on a new page is pre-allocated. No update of the page is performed while updating, only pre-allocation of space and update of a "volatile" memory region.

At commit time all pages involved are loaded into memory before starting the commit logic. Now the pages are updated and the space pre-allocated is committed.

As long as the "volatile" memory fits in main memory this algorithm suffers no problems. Large transactions require very large memory regions to contain the ongoing changes. To be able to handle transactions of any size, even the "volatile" memory can be spooled out on disk. This is performed on a per-transaction basis. When a transaction reaches a threshold size, all new updates of this transaction is using a special memory space that can easily be sent to disk in its completeness. When such a transaction is to commit, this memory must be read back into memory. At rollback this memory and the file connected to it can be deallocated and deleted. At recovery any such files can be deleted since they have no impact on the recovery logic.

8.2 INSERT algorithm

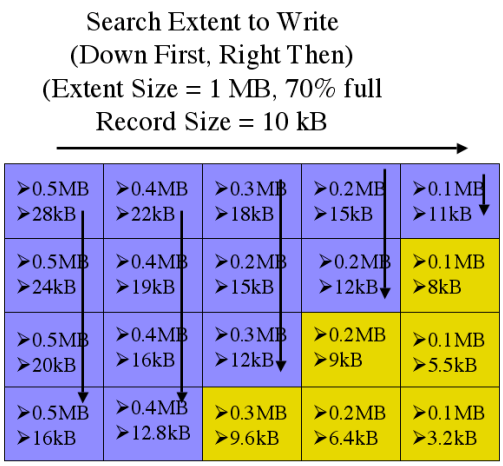
The difference with an INSERT is that it is mandatory to pre-allocate space on memory pages and disk pages for the record. This is performed at the end of the INSERT when the size of the record parts is known.

An intricate algorithm is used to find the optimum disk page to house the disk part. There are five steps in this algorithm. Each step is performed if possible and the next step is only done if the previous wasn't possible.

1) Find any page that fits the record in the current set of "dirty" pages in the page cache.
2) Find any page that fits the record in the current set of pages currently read from disk to be used in INSERT/UPDATE/DELETE.
3) Find a new page in the same extent that fits the record.
4) Find a new extent using a two-dimensional matrix.
5) Allocate a new extent.

The reasoning behind step 1) and 2) is that as much updated data as possible is written to disk when a page is flushed from the page cache. The reasoning behind step 3) is to use one extent and ensure that all new allocations as much as possible are done by sequential writes. New writes are grouped into writes larger than one page as much as possible to minimise cost of each write.

Step 4) is the most complex step and requires some careful explanation. The free space in all extents will in the long-run form a variant of the normal probability distribution. When a new extent is picked to write from it is desirable to find an extent that has as much free space as possible. At the same time it is necessary that it contain at least one page with enough space to fit our needs. The first dimension in the matrix is the minimum of the maximum free space on one page in the extent. The second dimension is the size of the free space in the extent. The search tries to find an extent that has the largest possible free space on a page and the largest free space in the extent. The figure shows the search order.



The boundary values in the matrix are calculated using the proper probability distribution such that an even distribution of extents is accomplished. This is future work to optimise the selection of these values.

Finally if no such extent is found, a new extent is allocated from the proper tablespace.

The discussion above is still only to preallocate area in a page so no change of the page is performed. A separate subtransaction to allocate a new extent might be started but this will commit even if the INSERT transaction is rolled back. The INSERT itself is first performed in a "volatile" area as for UPDATE's. Thus even abort of INSERT's are done independent of handling of recovery.

8.3 Reasoning behind data placement

For INSERT's the intention is to choose a page such that page writes occur on many sequential pages. It is never attempted to gather writes bigger than an extent.

For UPDATE's a possibility is to always write in a new place since we have the directory in the main memory. There is still however a need to update the free list of the page to ensure the space is reused. Since it is necessary to read the record to update it, it was deemed more efficient to do a in-place update if possible. If the updated record no longer fits in memory it is moved in its entirety to a new page. This follows the same procedure as an INSERT.

The in-place updates will require writing pages through random access. Thus it is known that this page is written at some point in time and thus it is prioritised to write more records on this page since there is no extra cost of writing this page with more records compared to only one record.

In future releases of MySQL Cluster there are ideas to move disk parts to "hotter" pages if often used (or even to main memory pages) and likewise to "colder" pages if not used for a longer period. This will greatly improve the cache hit rate in the page cache.

References

[1]Elmasri, Navathe. Fundamentals of Database Systems, Third Edition

[2]M. Ronström. High Availability Features of MySQL Cluster, White Paper MySQL AB 2005-02-18

[3]M. Ronström. Design of a Parallel Data Server for Telecom Applications, Ph.D. thesis, Linkoping University, 1998

[4]U. Troppens, R. Erkens, W. Müller. Storage Networks Explained, Wiley 2004

[5]J.Gray, A. Reuter. Transaction Processing: Concepts and Techniques, Morgan Kaufmann 1993

[6]W. Effelsberg, T. Haeder. Principles of Data Buffer Management, ACM TODS 9(4):560-595

[7]D. Skeen, A. Chan, The Reliability Subsystem of a Distributed Database Manager, Technical Report CCA-85-02, Computer Corporation of America, 1985

[8]S-O Hvasshovd, A Tuple Oriented Recovery Method for a continuously available distributed DBMS on a shared-nothing computer, Ph.D. Thesis, NTH Trondheim, 1992, ISBN 82-7119-373-2