Datalogisk Institut
KØBENHAVNS UNIVERSITET

My SQL in a Main Memory Database Context

Michael Lodberg Samuel and Anders Uhl Pedersen

Technical Report no. 2004/02
ISSN: 0107-8283

DIKU
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

MySQL in a Main Memory Database Context

Michael Lodberg Samuel
Anders Uhl Pedersen

Department of Computer Science
University of Copenhagen

Abstract

In main memory database systems (MMDB) the primary copy of data is stored in memory. This paper explores how main memory residency of data impacts data management, and examines MySQL 4.1 in this context. Suggestions on how to improve the main memory parts of MySQL are presented.

1 Introduction

As the cost of RAM decreases, it becomes more interesting to use databases in which all or a major portion of the database is placed in memory – Main Memory Database Systems. Due to the different access properties of RAM compared to harddisks, MMDBs have traditionally used changed or new storage- and access-strategies compared to disk-based database systems, but as cache-size and -levels increases, RAM's access properties change as well.

In this paper, we do a general discussion of some of the ways main memory database systems differ from disk resident database systems.

We then move on to do a more specific examination of MySQL 4.1 in this context, and provide suggestions on how to improve the main memory parts of MySQL in terms of performance and functionality.

2 Main memory database systems

In the following subsections we discuss how memory residency of data impacts database management in different functional areas.

2.1 Data Representation

When moving from disk resident databases (DRDB) to main memory databases (MMDB) priorities in how to store data shift. This is due to significant changes in price of storage and access speed. In DRDB it is widely accepted that using a little more space in order to gain efficiency in I/O is a good trade-off.

In MMDB, the size of data becomes more important as RAM tend to be more expensive and less voluminous than hard disk space. At the time of writing, an off-the-shelf 250GB hard drive can be purchased for app. €300 while 1GB of RAM will cost app. €200.

For DRDB-systems it has always been important to identify objects that are frequently accessed at the same time[1] and group these objects together on disk and in the buffer-manager in order to keep the relatively expensive I/O-waits to a minimum. Especially in early articles on the subject of MMDB [1,2] the removal of I/O-waits has diminished this concern, since RAM[2] by definition is more suited for random access patterns than disks are.

Also, with the increased speed of access through RAM compared to disk, focus has been on data-management algorithms, trying to decrease the amount of computations needed – even at the cost of wasted space in some cases - when accessing data, as this was perceived as a potential bottleneck.

In Starburst [1], the MM-component has been built on top of the existing DRDB-system. This means that the record format remains the same in both versions, as this seemed more efficient than doing conversions on-the-fly between the two formats. The DRDB-record structure that has self-relative offsets as pointers to the various fields in the record, was enhanced with a separate block (that can be placed anywhere in memory) containing pointers to the memory addresses of each field in the record. Apart from bypassing the buffer manager, the need to calculate actual addresses from the offsets is thereby eliminated.

Dali[3] [3,4] supports different level APIs and uses a heap file as the abstraction on the higher levels. On the low level, it gives the user direct access to the database files. In this scenario, the user will have responsibility for maintaining the correct pointers and moving data around if need be. The heap file only supports fixed-length items, so moving records shouldn't be an issue.

[1] One example could be fields in the same record.
[2] Random Access Memory
[3] Now called DataBlitz

In Starburst [1], the concept of tombstones is introduced when records grow. Tombstones are basically a way of being able to find a record that has been moved from its original location without looping through all MM-record structures to update pointers to it, by placing a pointer to a new address in the record's previous location.

The problem with this – and with the general celebration of the direct address-pointers as a fabulous improvement for data representation – is that memory caching has become increasingly important recently, meaning that random access patterns are not as effective for RAM as they have been. While the mechanical factors for disks (i.e. search time latency due to the relocation of the heads) are not an issue for RAM, the segmentation of RAM into main memory and several levels of faster (and more expensive) cache-RAM once again makes locality of data an issue.

This reintroduces some of the same problems that were present for DRDB-systems – and especially concepts such as tombstones and some of the indexing methods give severe performance problems with modern architectures, since they enforce a jumping between different areas in memory, which effectively reduce cache utilization [5].

2.2 Access Methods

The most notable change in access methods between MMDBs and DRDBs in general, is the lack of the buffer manager for MMDBs.

In DRDBs a buffer manager is present to provide efficient communication between memory and disk, keeping the hot data ready in memory while swapping the cold data out to disk. As all access in MMDBs is in memory, a buffer manager is not needed. And while the buffer managers are optimized with various algorithms for finding and keeping the right data ready at the right time, they themselves add computation time to accessing data – and the recalculation and mapping of disk-addresses to memory addresses increases usage of precious clock cycles.

When the buffer manager (and disk manager) is bypassed, the need for copying data from storage to buffer and mapping addresses accordingly is also eliminated.

This also affects indexes. Traditionally, they contain (some of) the data that they are indexing – in the case that these data will be sufficient, expensive lookups can be avoided. This approach has been considered irrelevant in MMDBs since pointer lookups are so fast [2] – and when only storing pointers in the indexes, the nodes will (usually) be able to hold more elements, making the tree smaller and decreasing the time it will take to traverse it. This has been an obvious benefit, since a lot of effort has gone into maximising node-capacity in DRDB-indexes.

However, with the increased importance of the cache, especially the indirect indexes see significant decreases in the competitive performance – simply because cache misses become predominant and end up taking more time than the actual looking up of data [5].

T-trees are basically balanced binary trees, but with many key-elements in each node. They have a fairly good space overhead, although they also store a record pointer for each key in the node, but they have a poor cache performance, especially with indirect pointers [5,6].

B+ trees only store keys and pointers to child nodes (how many of these that will exist is a matter of individual design) in the internal nodes. Only leaf nodes have record pointers. This means that it will always be necessary to traverse to the bottom of the tree in order to look up a value. The leaves are joined together in a linked list, making range scans very efficient. A cache sensitive variant (CSB+-tree) has been proposed [7].

Hashing can be implemented in a number of variations with many different strategies. Generally, hashing is fast for point searches, but lookups have to be done over and over if a range scan needs to be carried out, as two elements that would be adjacent in a tree will most likely not be placed in the same bucket.

Various optimizations concerning cache-sizes or their multiples can be considered for all of the index methods [6,7,8].

2.3 Query processing

When doing query processing in a MMDB rather than a DRDB, *CPU computation time* and (increasingly) *cache utilization* has replaced *number of disk I/O operations* as the dominant cost. This affects query processing in a number of ways.

It is e.g. more efficient to replace foreign key values (e.g. departmentID in an Employee relation) with foreign key pointers. This way a join consists of pointer comparisons instead of value comparisons.

The potential cost savings can be significant if the keys consist of string values.

Similarly, a materialization of intermediate results only needs to contain pointers to the actual values, i.e. no data need to be copied to the intermediate table.

In MMDB join methods that take advantage of faster sequential access[4] loose some appeal not least because of the overhead in terms of CPU-processing and space usage imposed by sorting relations before joining. While older experimental results have confirmed that T-tree based merge joins usually outperform sort-merge joins [9], more recent results indicate that sort-merge join still has a place in main memory if the sorting method is cache conscious [10].

2.4 Concurrency control

In the absence of disk accesses and possibly due to pre-committing, transaction processing in main memory is much faster, which implies that locks are held for a smaller amount of time. This suggests that coarse granularity locking such as table level locking is feasible, thereby avoiding much of the locking management overhead in current DRDBs. However, this reduces protection from long-running transactions causing delays or from starvation and a more flexible locking protocol should be considered. For instance a locking protocol that automatically switches from coarse to fine granularity whenever major lock contention appears or a long-running transaction participates in lock conflicts. However, to accomplish this de-escalation of locks it is necessary that the lock manager keeps track, not only of current coarse locks, but also of any potential fine granularity locks that could be required.
In DRDBs a hash-table is the typical lock management data structure. In a highly concurrent environment that supports fine granularity locks such a hash-table could become a serious bottleneck. In [1] the problem is circumvented by attaching lock information to data itself. Since de-escalation of locks is supported, the table lock managers also keep track of all tuple lock requests so far. De-escalation of locks consists of deactivating the table lock manager and activating all the local tuple lock managers by applying the tuple lock information in the table lock manager to the tuple lock managers.

[4] I.e. sort-merge

2.5 Commit processing

When a transaction commits its activity log records must be flushed to stable storage in order to ensure transaction durability. If non-volatile memory is not present then the log data must be flushed to disk, which makes disk I/O a likely bottleneck in transaction processing.

One way of reducing this I/O bottleneck is to take advantage of the fact that only the redo log records have to be stored on stable storage[5]. Some recovery algorithms [11,12] exploit this by storing only *redo* log records in the system log while the *undo* log records are stored in a "local" log, i.e. in transaction buffer space. Only if the transaction fails, the undo log records must be written to the system log. This reduces the amount of log data that must be flushed during commit processing.

The idea of a local transaction buffer can be further utilized. Some recovery algorithms [12] use a shadowing technique instead of in-place updating. This means that all the updates are performed on local copies of data. When the transaction commits the local updates are transferred to the primary memory database. If a transaction fails then the local updates are just discarded and no undo log records are required.

Using group commit can further reduce the disk I/O bottleneck. That is, to delay the flushing of the log records for a committing transaction until more transactions are ready to commit. This increases the amount of log data to be flushed at once and reduces the number of I/O operations, but increases the average commit processing time slightly [12].

Pre-committing does not reduce the I/O bottleneck but it can increase concurrency. Pre-commit is the idea that a transaction is allowed to release its locks as soon as its log records have made it to the log in memory. This means that a commit-ready transaction does not delay the execution of other transactions while its log records are flushed to disk [11,13].

If pre-commiting is not enabled when using group commit, group commit will also have a detrimental effect on concurrency.

If stable memory is present then the I/O bottleneck is less critical because log flushing can be decoupled from commit processing. When the redo log records

[5] Undo log records might be required if checkpointing

have made it to stable memory the transaction has committed, and the log records can subsequently be flushed to disk in a more efficient manner [14,15].

2.6 Checkpointing

Fast recovery highly depends on the existence of a nearly up-to-date checkpoint. On the other hand checkpointing can significantly impact database processing – especially in a large memory context. A strategy that halts transaction executing while checkpointing is running is generally unwanted. In MMDBs that will often be tuned for real-time processing, it is unacceptable. This favours fuzzy checkpointing methods [13,16] and strategies that only flush dirty pages [15].

If log data for active transactions are not stored in stable memory a fuzzy checkpoint may violate the WAL protocol because the undo log record corresponding to an update in a checkpointed page, may not yet have been flushed to disk. One way of dealing with this problem is to keep the previous checkpoint image of the database on disk while checkpointing [13].

To save disk space a segmented fuzzy checkpointing strategy has been proposed [16] which checkpoints only a segment of the database at a time in a round-robin fashion. The segment boundaries are automatically changed based on the distribution of update-operations. A smaller amount of the log must be read during recovery with this approach.

2.7 Recovery

In case of system or media failure the database is lost and its most recent consistent state must be constructed in memory using the copy of data and log on stable storage.

Simple recovery methods do not allow transaction execution to resume until the entire database is present in memory in a consistent state [12]. Obviously, this is undesirable and a concurrent reloading strategy is more convenient because transaction processing can resume even before the database has been reloaded to a consistent state [4,14,15].

Some concurrent reloading strategies allow transaction processing to be performed as soon as the system catalogues and their indices have been reloaded [17].

If a transaction requests data not yet in memory the transaction manager will forward the request to the recovery manager and the data must be copied to memory.

One way of doing this is to only copy the requested data back into memory if the data resides in a disk page containing merely committed data [15]. If it is not the case the missing page updates are executed and the page is loaded into memory. For this strategy to be efficient it requires grouping log records on a page basis during normal processing.

Another approach is to copy the requested page into memory even if it contains uncommitted data. However, before transaction processing can resume the logged after images - of pages updated since the most recent checkpoint - are loaded into memory [18]. When a new transaction requests data on some page it checks if an after image version exists. If so the after image version takes precedence over the just loaded version from disk.

3 MySQL

MySQL has recently acquired Alzato [19], a company developing a clustered main memory database management system (MMDB). It is designed for applications that require maximum uptime and real-time performance, such as telecom and network applications and heavy-load websites. In this clustered MMDB a table is divided into fragments distributed among the cluster nodes. The nodes are combined into node groups and each fragment has all its replicas in one node group. Replicas are needed since data is stored in main memory at each node. A transaction has committed when all replica nodes have (synchronously) updated the log and the memory copy of data. Only subsequently are the committed updates propagated to disk. This means that even committed data might be lost if all members of a node group are lost.

The rest of the article will be devoted to a discussion of the heap tables in MySQL.

MySQL internally uses heap tables as temporary tables located in memory. If the temporary table becomes too big it is automatically converted to a disk table. Heap tables are also externally available, enabling the user to create memory resident tables. However, these are not automatically moved to disk in case the table no longer fits into the available RAM. Since recovery routines are not implemented heap tables are currently only recommendable for holding temporary data or copies of data already stored on disk. The heap table could e.g. provide a nearly up-

to-date copy of a disk table (which can then record the changes), making data available for fast access. The heap table could be regenerated from the disk table e.g. once a day.

3.1 Data representation

A record in the heap table consists of a fixed-size key part followed by the fixed-size fields of the record. Since the record length is fixed an update cannot cause space overflow, which means that a record never needs reallocation. This eliminates the need of e.g. concepts like the tombstones that are used in Starburst to deal with reallocation. Also, the space occupied by a record is not de-allocated on record deletion. The free record slot is just appended to a list of free slots and is reused in case of subsequent inserts.

The records can be accessed through a tree structure containing records at the leaf level. Each inner node (also called a block) contains up to 128 pointers so the node size is optimised for today's typical cache line size.

On insertion of a record a position number is attached to the new record as if all the record pointers were stored in a simple array. This "array" position number is translated to the matching position in a leaf node in the tree structure by traversing down the tree. The insertion may require allocation of a new leaf node and possibly up to four inner nodes[6], depending on the current height of the tree. With a leaf node size of approximately 65KB the number of records stored in a leaf depends on the record size.

Unfortunately the leaf nodes are not linked, and due to implementation decisions it is not possible to move freely around in the tree which means that a table scan or a range scan requires traversing the tree from root to leaf every time the next leaf node must be read. Also, since the access methods currently implemented are severely limited, table scans are not well supported in the current implementation.

3.2 Access methods

At present the only externally available index is a hash index. As the index is indirect, a bucket element consists of *a pointer to* the key part of the record and a pointer to the next element in the same bucket.

The hash table is based on the same tree structure mentioned above and it is incremental in nature. On insertion of an element the record key is mapped to a hash key, which in turn is mapped to a bucket number x (0< x <= number of inserted records+1). This bucket number is equivalent to the "array" position number used for records above and the bucket number is in turn translated to an actual leaf position in the tree structure.

If no bucket element is stored at this leaf position then the element is just inserted. However, if an existing element is already stored at this location it is moved to a new free leaf position (if necessary a new leaf node is allocated). At this time the element to be inserted can be stored at the original, now unoccupied, position. If the hash key of the moved element mapped to the same bucket number as the hash key of the inserted element then it is part of the same bucket so the next-element-in-bucket pointer of the inserted element must point to the moved element.

As described below, the hash table is special in that elements can be placed in different buckets depending on the size of the allocated space. Among other things, this means that an item in a bucket might need to be replaced after the space grows or shrinks.

The algorithm above works because special care is taken when mapping a hash key to a bucket number.
If the number of inserted records consists of n bits then the bucket number is the n least significant bits of the hash key. However, the most significant bit of this n-bit bucket number is zeroed in case the bucket number is larger than the number of inserted records. If the number of inserted records at some point changes to an n+1 bit value then the bucket number is equally computed using the n+1 least significant bits of the hash key. This situation needs special care as some of the already inserted elements in the "lower half" (n bit bucket numbers) now map to a new "upper half" (n+1 bit) bucket number if the n+1 bit of the hash key is 1. Each time we insert an element we check if some elements (stored in the next unchecked lower half bucket) must be moved to the corresponding upper half bucket. Notice that moving elements to the upper half bucket only requires physically moving at most one element to the new leaf position. Apart from this only the next-element-in-bucket pointers of the elements must be updated.

Because of the incremental insertion algorithm the lookup procedure has to search both in the lower half bucket and in the upper half bucket of the hash table because some elements might not yet have been moved to the upper half at the time of lookup.

[6] The maximum height of the tree is 5

At deletion of an element the number of inserted records is decremented. As a result the maximal bucket number is decremented as well. Consequently the element that is stored in the leaf position corresponding to the old maximal bucket number must be moved to another leaf position. The lower half bucket number to which its hash key now maps determines to which leaf position it must be moved. Finally the element at this position is moved to the (empty) leaf position of the originally deleted element.
When the number of inserted records after a series of deletions is reduced to an n-1 value all elements are consequently stored in leaf positions corresponding to lower half bucket numbers and the bucket numbers can accordingly be computed using only the n-1 bits of the hash key. The upper half space could potentially be de-allocated but this is not done.

If the key value of a record is updated it is obviously necessary to reinsert the element containing the record key pointer in each index as it probably maps to a different bucket.

3.3 Query processing

Heap tables are used internally as temporary tables to store intermediate results when processing complex queries. The query processor treats heap tables almost like disk based tables in the sense that only join order and access method are affected. Since hashing is the only available index method, the query optimizer has no range estimates, i.e. information on how many rows there are between two values and this might affect the decision on which index to use. Furthermore the indexes do not support ORDER BY.

3.4 Concurrency control, commit processing, checkpointing and recovery

Table level locking is the only option for heap tables. This is sufficient, as commit processing and long-running transactions are not a concern since transaction processing is not supported. Heap tables are purely memory resident, which means that recovery and checkpointing are not provided.

4 Recommendations

Currently MySQL is not competitive when compared to real main memory database systems since heap table recovery is not implemented. Nevertheless the functionality and performance of the heap tables can be improved in a number of ways, some of which will be discussed below.

The hash index is designed in a way that allows the number of buckets to be equal to the number of inserted elements. To minimize the number of cache misses pr. lookup the number of elements in each bucket must be minimized, since the elements in a bucket are not grouped in memory. An ideal hash function would consequently lead to only 1-element buckets, which is why the hash function is crucial for the performance of the hash index. It might be beneficial to reconsider the choice of hash function through a performance study.

Finding the next element in a bucket will likely result in a cache miss as mentioned above. As the hash index is indirect the actual key is not stored in the bucket element, only a pointer to the location of the key in the record is stored. Fetching the key thus results in an extra cache miss. A direct-key approach would most likely shorten the lookup time especially because the fixed key size simplifies such an approach. One might justly argue that storing the key values in the index is more space consuming and that for large key sizes it might be acceptable to use an indirect index.

Currently only a fixed-length record format is supported which comprises fixed-size keys. Obviously the utilization of the heap tables is enhanced if variable-length fields are supported. While variable-length *records* introduce some allocation issues (overflow, fragmentation etc.), variable-length *keys* make direct-key indexing more complex. To support variable-length keys and utilize the current hash index algorithm a partial-key approach seems to be the most practical option. Partial-key indexing might also be worthwhile for long keys in order to limit space consumption [5].

While the benefit of de-allocating free space in the hash index is limited, de-allocation of unused record space due to record deletions is worth consideration. Reaching a certain watermark could e.g. trigger a de-allocation background thread. As memory for record storage is allocated in chunks of 65KB an undemanding approach would be to de-allocate only empty chunks. This way de-fragmentation stays out of the picture. However, de-fragmentation might improve cache utilization so this is a trade-off.

If the query optimizer prefers a table scan the record tree structure comes into play. As mentioned a table scan requires traversing the tree from root to leaf every time the next leaf node must be read, as the leaf

nodes are not linked. One improvement could be to join together the leaves in a linked list.

The traversal of the tree can be improved as well. With a leaf node of approximately 65KB the number of records stored in a leaf node depends on the record size. As previously mentioned the number of pointers stored in inner nodes is 128, i.e. a power of 2. If the number of records in a leaf node is adjusted to be a power of 2 as well then the tree can be traversed using bit shifting, thereby eliminating the expensive modulo and division computations.

In general the heap table implementation relies heavily on pointer arithmetic even in cases where simple array indexing would suffice. Rewriting pointer references to direct references might lead to a general speed up because it assists modern compilers in identifying what to prefetch. It would also make the system easier to maintain by improving readability of the code.

References

1. Lehman et al.: „Evaluation of Starburst's Memory Resident Storage", *IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6*, December 1992.

2. Garcia-Molina & Salem: "Main Memory Database Systems: An Overview", *IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6*, December 1992.

3. P. Bohannon, D. Lieuwen, R. Rastogi, A. Silberschatz, S. Seshadri and S. Sudarshan, "The Architecture of the Dali Main-Memory Storage Manager", *Multimedia Tools and Applications, Vol. 4, No. 2*, 1997

4. "DataBlitz Architectural Overview", http://citeseer.nj.nec.com/315749.html

5. P. Bohannon, P. McIlroy and R. Rastogi, "Main-Memory Index Structures with Fixed-Size Partial Keys", *ACM SIGMOD Conference*, 2001.

6. J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory", *The VLDB Journal*, 1999.

7. J. Rao and K. A. Ross, Making B + -Trees Cache Conscious in Main Memory", *SIGMOD Conference*, 2000.

8. G. Graefe, R. Bunker, and S. Cooper, "Hash joins and hash teams in Microsoft SQL server", VLDB, 1998.

9. T. Lehman and M.J. Carey, "Query Processing in Main Memory Database Management Systems", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1986.

10. C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray and D. Lomet, "AlphaSort: A RISC machine sort", *In SIGMOD*, 1994.

11. H. V. Jagadish, A. Silberschatz and S. Sudershan, „Recovering from Main-Memory Lapses", *VLDB*, 1993.

12. E. Levy and A. Silberschatz, "Log-Driven Backups: A recovery Scheme for Large Memory Database Systems", *The 5th Jerusalem Conf. on Information Technology (JCIT)*, 1990.

13. P. Bohannon, R. Rastogi, A. Silberschatz and S. Sudarshan, „Multi-Level Recovery in the Dalí Main-Memory Storage Manager", 1998

14. Y. Wang and V. Kumar, "A Main Memory Recovery Algorithm with no Checkpointing and its Performance Analysis", *Data & Knowledge Engineering*, submitted for publication, 1997

15. E. Levy and A. Silberschatz, "Incremental recovery in main memory database systems", *IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6*, 1992.

16. J. Lin and M. H. Dunham, "Segmented fuzzy checkpointing for main memory databases", *Selected Areas in Cryptography*, 1996.

17. T. Lehman and M. J. Carey, "A recovery algorithm for a high performance memory-resident database system", *In 19 ACM SIGMOD Conf. on the Management of Data*, 1987.

18. Le Gruenwald and Margaret H. Eich, "MMDB reload algorithms", 1991.

19. Alzato, http://www.alzato.com