Main Memory Database Systems: An Overview

Hector Garcia-Molina, *Member, IEEE*, and Kenneth Salem, *Member, IEEE*

Invited Paper

***Abstract*—Memory resident database systems (MMDB's) store their data in main physical memory and provide very high-speed access. Conventional database systems are optimized for the particular characteristics of disk storage mechanisms. Memory resident systems, on the other hand, use different optimizations to structure and organize data, as well as to make it reliable. This paper surveys the major memory residence optimizations and briefly discusses some of the memory resident systems that have been designed or implemented.**

***Index Terms*— Access methods, application programming interface, commit processing, concurrency control, data clustering, data representation, main memory database system (MMDB), query processing, recovery.**

I. Introduction

In a *main memory database system* (MMDB) data resides permanently in main physical memory; in a conventional database system (DRDB) it is disk resident. In a DRDB, disk data may be cached into memory for access; in a MMDB the memory resident data may have a backup copy on disk. So in both cases, a given object can have copies both in memory and on disk. The key difference is that in MMDB the primary copy lives *permanently* in memory, and this has important implications (to be discussed) as to how it is structured and accessed.

As semiconductor memory becomes cheaper and chip densities increase, it becomes feasible to store larger and larger databases in memory, making MMDB's a reality. Because data can be accessed directly in memory, MMDB's can provide much better response times and transaction throughputs, as compared to DRDB's. This is especially important for real-time applications where transactions have to be completed by their specified deadlines.

A computer's main memory clearly has different properties from that of magnetic disks, and these differences have profound implications on the design and performance of the database system. Although these differences are well known, it is worthwhile reviewing them briefly.

1) The access time for main memory is orders of magnitude less than for disk storage.
2) Main memory is normally volatile, while disk storage is not. However, it is possible (at some cost) to construct nonvolatile main memory.
3) Disks have a high, fixed cost per access that does not depend on the amount of data that is retrieved during the access. For this reason, disks are block-oriented storage devices. Main memory is not block oriented.
4) The layout of data on a disk is much more critical than the layout of data in main memory, since sequential access to a disk is faster than random access. Sequential access is not as important in main memories.
5) Main memory is normally directly accessible by the processor(s), while disks are not. This may make data in main memory more vulnerable than disk resident data to software errors.

These differences have effects on almost every aspect of database management, from concurrency control to application interfaces. In this paper we will discuss these effects and will briefly survey some MMDB's that have been recently designed or implemented. However, before getting started, it is important to address three questions that are commonly asked about MMDB's.

Is it reasonable to assume that the entire database fits in main memory? Yes, for some applications. In some cases, the database is of limited size or is growing at a slower rate than memory capacities are growing. For example, the database size may be proportional to the number of employees or customers in a company, and no matter how successful the company is, it is reasonable to expect that memory can hold a few hundred or thousand bytes per employee or customer. In some real time applications, the data *must* be memory resident to meet the real-time constraints, so the database will necessarily be smaller than the amount of available memory. Example of such real-time applications include telecommunications (e.g., 800 telephone numbers need to be translated to real numbers), radar tracking (e.g., signatures of objects need to be matched against a database of known aircraft), and securities trading (e.g., trading opportunities must be discovered and executed before they vanish).

However, there are clearly cases where the database does not and will never fit in memory, e.g., an application with satellite image data. For such cases, DRDB will continue to be important. Nevertheless, even in these very large applications, it is common to find different classes of data: *hot* data that is accessed frequently, usually low volume and with stringent timing requirements, *cold* data that is accessed rarely and is more voluminous, and various intermediate degrees. If this is

Manuscript received December 1, 1991; revised July 27, 1992. The work of K. Salem was supported by the National Science Foundation under Grant CCR-8908898 and by CESDIS.

H. Garcia-Molina is with the Department of Computer Science, Stanford University, Stanford, CA 94305.

K. Salem is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

IEEE Log Number 9204082.

the case, it is possible to partition the data into one or more logical *databases*, and to store the hottest one in main memory. We then have a collection of databases, some managed by a MMDB, others by a DRDB. The database systems may be totally disjoint, so that applications access them in much the same way they would access a lose federation of database systems; or they may be tightly integrated, with facilities for automatic data migration from one database system to the other, as the access frequency of the data changes. Note that this migration is not simply caching of values. With caching, a temporary copy of an object is made at a different level of the storage hierarchy. With migration, the object moves to a different management system, and its structure and access mechanisms may change.

There are many applications where this partition of data arises naturally. For example, in banking, account records (e.g., containing balances) are usually hot; customer records (e.g., containing address, mother's maiden name) are colder. Historical records (showing banking activities) are usually cold. In a telephone switching application, routing tables (e.g., mapping 800 phone numbers to actual numbers) are hot; data for customers' monthly statements are cold.

IMS, one of the earliest database systems, recognized these access differences, and has provided two systems in one for many years: Fast Path [9] for memory resident data, and conventional IMS for the rest. A recent paper by Stonebraker [25] also discusses some of the issues involved in multilevel database systems and data migration. In the rest of this paper, when we refer to "the database" we will be referring to that fraction of the total data that is permanently in memory and managed by the MMDB.

What is the difference between a MMDB and a DRDB with a very large cache? If the cache of a DRDB is large enough, copies of the data will reside in memory at all times. Although such a system will perform well, it is not taking full advantage of the memory. For example, the index structures will be designed for disk access (e.g., B-trees), even though the data are in memory. Also, applications may have to access data through a buffer manager, as if the data were on disk. For example, every time an application wishes to access a given tuple, its disk address will have to be computed, and then the buffer manager will be invoked to check if the corresponding block is in memory. Once the block is found, the tuple will be copied into an application tuple buffer, where it is actually examined. Clearly, if the record will always be in memory, it is more efficient to refer to it by its memory address.

What we have illustrated is only one of the possible in-memory optimizations. Others will be described in Section II. It is important to note that some DRDB and some object-oriented storage systems (OOSS) are beginning to recognize that with large caches some of their data will reside often in memory, and are beginning to implement some of the in-memory optimizations of MMDB. For example, some new systems convert a tuple or object into an in-memory representation and give applications a direct pointer to it. (This is called "swizzling" [4], [25].) As DRDB perform more and more in-memory optimizations, they become closer to MMDB. In the future, we expect that the differences between a MMDB and DRDB will disappear: any good database management system will recognize and exploit the fact that some data will reside permanently in memory and should be managed accordingly.

Can we assume that main memory is nonvolatile and reliable by introducing special purpose hardware? It is tempting to make this assumption since the design of a MMDB would be further simplified and performance would improve further (no crash recovery code at all!). There is no "yes" or "no" answer here. Memory is simply a storage medium that can be made more reliable by techniques such as battery-backed up memory boards, uninterruptable power supplies, error detecting and correcting memory, and triple modular redundancy. However, this only reduces the probability of media failure, but does not make it to zero. Thus one will always have to have a backup copy of the database, probably on disk. Note that for DRDB, backups are also required, possibly to tape or other disks.

As the probability of media failure decreases, the frequency of backups can decrease, and the performance implications of these backups decreases. For example, with good disks, it may be sufficient to back them up to tape once a week. The overhead of scanning and copying the entire database once a week (probably concurrently with other activities) should not be significant. For the case of memory resident data, there are several factors that force the frequency of backups up.

1) Since memory is directly accessible by the processor (see item 5), Section I), it is more vulnerable to operating system errors. So even if the hardware is very reliable, the contents of memory will be periodically lost when the system "crashes."
2) When one disk fails, it can be fixed without affecting the contents of other disks. After recovery, only a fraction of the database (on that disk) must be restored from the backup (and logs). Also, during recovery, the rest of the database may still be accessible. When a memory board fails, typically the entire machine must be powered down, losing the entire database. Since recovery of the data will be much more time consuming, it is desirable to have a recent backup available. (The older the backup, the more it takes to bring up to date from the log.)
3) Battery backed memory, or uninterruptable power supplies (UPS) are "active" devices and lead to higher probability of data loss than do disks. Disks are "passive" and do not have to do anything to remember their data. An UPS can run out of gas or can overheat. Batteries can leak or lose their charge.

In summary, unless one has a lot of trust in memory reliability, memory backups will have to be taken relatively frequently, and the performance of the backup mechanism will be of central importance. Also, because the cost of writing to disk is so much higher than that of writing to memory, the overhead of the backups will be much more significant than for the equivalent disk to tape backups in a conventional system (see Section II-G).

II. Impact of Memory Resident Data

We have argued that a database system should manage memory resident data differently from a conventional system.

In the following subsections, we discuss the impact of memory residency on some of the functional components of database management systems.

A. Concurrency Control

Because access to main memory is so much faster than disk access, we can expect transactions to complete more quickly in a main memory system. In systems that use lock-based concurrency controls, this means that locks will not be held as long, and suggests that lock contention may not be as important as it is when the data is disk resident. (Here we focus on locking concurrency control since it is the most commonly used in practice and is what has been used in MMDB prototypes. However, we expect that optimizations similar to the ones to be described could be used for other types of mechanisms, such as optimistic or time-stamp based.)

Systems that choose small locking granules (fields or records) do so to reduce contention. If contention is already low because data are memory resident, the principal advantage of small lock granules is effectively removed. For this reason, it has been suggested that very large lock granules (e.g., relations) are most appropriate for memory resident data [17].

In the extreme, the lock granule could be chosen to be the entire database [8], [18]. This amounts to serial execution of transactions. Serial transaction processing is highly desirable, since the costs of concurrency control (setting and releasing locks, coping with deadlock) are almost completely eliminated. Furthermore, the number of CPU cache flushes is greatly reduced. (Each time a transaction is suspended waiting for a lock, a new transaction is run and the contents of the CPU cache must change. With serial execution, only one flush needs to occur per transaction.) In high performance computers, where cache flushes are equivalent to thousands of instructions, the gains can be very significant. However, serial transactions are probably not practical when long transactions (e.g., conversational transactions) are present. For fairness, there should be some way to run short transactions concurrently with long-lived ones. Furthermore, multiprocessor systems may require some form of concurrency control even if all transactions are short.

The actual implementation of the locking mechanism can also be optimized for memory residence of the objects to be locked. In a conventional system, locks are implemented via a hash table that contains entries for the objects currently locked. The objects themselves (on disk) contain no lock information. If the objects are in memory, we may be able to afford a small number of bits in them to represent their lock status.

To illustrate, say we are dealing with exclusive locks only. (We are told IMS uses this idea, although we have not found a published reference to it.) If the first bit is set, then the object is locked, else it is free. If it is locked and the second bit is set, then there are one or more waiting transactions. The identity of these waiting transactions is stored in a conventional hash lock table. If a transaction wishes to lock an object, it first checks its lock bit. If it is not set, it sets it and is done with the locking process. (Some type of test and set instruction must be used to avoid two transactions from setting the bit.) Later on, if a second transaction wants to wait on the object, it sets the second bit on and adds itself to the list of waiting transactions in the lock table.

When the original transaction releases its lock bit, it checks if the second bit is set. If not, there are no waiting transactions and it is done. If it is set, it must go through the conventional procedure to wake up a waiting transaction. Clearly, we have omitted many details. However, the key point is that by far the most likely situation (with low contention) is for a transaction to lock a free object, update it, and to release its lock before any other transaction waits for it. In this case, both the lock and the release can be done with a minimal number of machine instructions, avoiding the hash table lookup entirely. Of course, there is the extra space overhead to consider. However, for "typical" database records that are tens of bytes or more long, the overhead of two bits may not be significant.

B. Commit Processing

To protect against media failures, it is necessary to have a backup copy (see Section I) and to keep a log of transaction activity. Since memory is usually volatile, this log must reside in stable storage (e.g., redundant disks). Before a transaction can commit, its activity records must be written to the log [11].

The need for a stable log threatens to undermine the performance advantages that can be achieved with memory resident data. Logging can impact response time, since each transaction must wait for at least one stable write before committing. Logging can also affect throughput if the log becomes a bottleneck. Although these problems also exist when data is disk resident, they are more severe in main memory systems because the logging represents the only disk operation each transaction will require.

Several solutions have been suggested for this problem. First, a small amount of stable main memory can be used to hold a portion of the log [5]–[8], [13], [17]. A transaction is committed by writing its log information into the stable memory, a relatively fast operation. A special process or processor is then responsible for copying data from the stable memory to the log disks. Although stable memory will not alleviate a log bottleneck, it can eliminate the response time problem, since transactions need never wait for disk operations. Studies have suggested that only a small amount (e.g., fewer than one hundred log pages [3]) of stable memory is needed to hold the log tail, even in high performance systems.

In case stable memory is not available for the log tail, transactions can be precommitted [5], [9]. Pre-committing is accomplished by releasing a transaction's locks as soon as its log record is placed in the log, without waiting for the information to be propagated to the disk. The sequential nature of the log ensures that transactions cannot commit before others on which they depend. Although precommitting a transaction does not reduce its response time, it may reduce the blocking delays (and hence, the response time) of other, concurrent transactions.

A technique called group commits can be used to relieve a log bottleneck [5], [9]. Under group commit, a transaction's log record need not be sent to the log disk as soon as it com-

mits. Instead, the records of several transactions are allowed to accumulate in memory. When enough have accumulated (e.g., when a page is full), all are flushed to the log disk in a single disk operation. Group commit reduces the total number of operations performed by the log disks since a single operation commits multiple transactions.

C. Access Methods

In a main memory database, index structures like B-Trees, which are designed for block-oriented storage, lose much of their appeal. A wide variety of index structures have been proposed and evaluated for main memory databases [5], [16], [26]. These include various forms of hashing and of trees. Hashing provides fast lookup and update, but may not be as space-efficient as a tree, and does not support range queries well. Trees such as the T-Tree have been designed explicitly for memory-resident databases [16]. Main memory trees need not have the short, bushy structure of a B-Tree, since traversing deeper trees is much faster in main memory than on a disk.

One observation common to all main memory access methods is that the data values on which the index is built need not be stored in the index itself, as is done in B-Trees. Because random access is fast in main memory, pointers can be followed quickly. Therefore, index structures can store pointers to the indexed data, rather than the data itself. This eliminates the problem of storing variable length fields in an index and saves space as long as the pointers are smaller than the data they point to.

The use of pointers suggests perhaps the simplest way to provide an index, which is simply to invert the relation on the indexed field [1], [2], [26]. In a main memory database, the inverted "relation" can simply be a list of tuple pointers in sorted order. Such indexes are very space efficient and are reasonably fast for range and exact-match queries, although updates are slow.

D. Data Representation

Main memory databases can also take advantage of efficient pointer following for data representation. Relational tuples can be represented as a set of pointers to data values [20], [26]. The use of pointers is space efficient when large values appear multiple times in the database, since the actual value needs to only be stored once. Pointers also simplify the handling of variable length fields since variable length data can be represented using pointers into a heap [17], [24].

E. Query Processing

Since sequential access is not significantly faster than random access in a memory resident database, query processing techniques that take advantage of faster sequential access lose that advantage [2], [5], [15], [20], [26]. An example is sort-merge join processing, which first creates sequential access by sorting the joined relations. Although the sorted relations could be represented easily in a main memory database using pointer lists, there is really no need for this since much of the motivation for sorting is already lost.

When relational tuples are implemented as a set of pointers to the data values (as discussed in Section II-D), some relational operations can be performed very efficiently [20]. For example, say we want to join relations R and S over a common attribute A. To perform the join we can scan the smaller relation, say R. For each tuple, we follow its A pointer to the actual value, call it a_i. From that value we follow back pointers to all S tuples that use a_i. We join the original R tuple to these S tuples, and add them to the result. For this to work, we need to have enough pointers in a_i to efficiently lead us to the S tuple that uses this value for its A attribute. Some additional storage will be required for this, but as is discussed in [20], the performance gains can be significant. The key idea is that because data is in memory, it is possible to construct appropriate, compact data structures that can speed up queries.

Query processors for memory resident data must focus on processing costs, whereas most conventional systems attempt to minimize disk access [26]. One difficulty is that processing costs can be difficult to measure in a complex data management system. Costly operations (e.g., creating an index or copying data) must first be identified, and then strategies must be designed to reduce their occurrence. Operation costs may vary substantially from system to system, so that an optimization technique that works well in one system may perform poorly in another.

F. Recovery

Backups of memory resident databases must be maintained on disk or other stable storage to insure against loss of the volatile data. Recovery has several components, the first being the procedure used during normal database operation to keep the backup up-to-date, and the second being the procedure used to recover from a failure.

We have already discussed commit processing, which is used to make sure that the results of all committed transactions are stable. Most systems that use a log for commit processing also perform backups or checkpoints to limit the amount of log data that must be processed to recover from a failure [5]–[7], [13], [17], [18], [23], [24]. Checkpointing brings the disk resident copy of the database more up-to-date, thereby eliminating the need for the least recent log entries.

In a memory resident database system, checkpointing and failure recovery are the only reasons to access the disk-resident copy of the database. Application transactions never require access to the disk resident data. Therefore, disk access in a memory resident system can be tailored to suit the needs of the checkpointer alone. One observation is that disk I/O should be performed using a very large block size. Large blocks are more efficiently written, and though they take longer, only the checkpointer (the system), and not the application transactions, await the completion of those writes.

Checkpointing should interfere as little as possible with transaction processing. Transaction-consistent or action-consistent checkpoints require some synchronization (e.g., locking) with transactions. An alternative known as fuzzy dumping requires no synchronization. However, consistent

checkpoints may simplify logging, since logical operations can be logged.

After a failure, a memory resident database manager must restore its data from the disk resident backup and then bring it up to date using the log. If the database is large, simply transferring the data from the disks may take a long time. One possible solution to this problem is to load blocks of the database "on demand" until all of the data has been loaded [12], [17]. However, it is not clear how much of an improvement this will provide in a high-performance system which must handle the demands of thousands of transactions in the seconds after the database has recovered.

Another possible solution to the database restoration problem is to use disk striping or disk arrays [14], [19], [22]. Here the database is spread across multiple disks, and it is read in parallel. For this to be effective, there must be independent paths from the disks to memory.

G. Performance

Performance is a concern for each of the database components we have described. With the possible exception of commit processing, the performance of a main memory database manager depends primarily on processing time, and not on the disks. Even recovery management, which involves the disks, affects performance primarily through the processor, since disk operations are normally performed outside the critical paths of the transactions. Most performance analyses of main memory techniques reflect this and the model processing costs [6], [17], [23]. This contrasts with models of disk-based systems (e.g., [21]) which count I/O operations to determine the performance of an algorithm.

Not only are the metrics different in MMDB analysis, but the MMDB components under analysis tend to be different. For example, in conventional systems, making backups (i.e., taking checkpoints) does not impact performance during normal system operation, so this component tends not to be studied carefully. As we argued in Section I, in a MMDB, backups will be more frequent and will involve writes to devices an order of magnitude slower than memory. Thus the performance of backup or checkpointing algorithms is much more critical and studied more carefully (e.g., [24]).

H. Application Programming Interface and Protection

In conventional DRDB's, applications exchange data with the database management system via private buffers. For instance, to read an object, the application calls the database system, giving the object id and the address of a buffer in its address space. The system reads the object from the disk into its own buffer pool and then copies the object to the application's private buffer. To write, the application modifies the private buffer, and calls the system. The system copies the modified object back to its buffer pool, and makes the corresponding log entries and I/O operations.

In a MMDB, access to objects can be more efficient. First, applications may be given the actual memory position of the object, which is used instead of a more general object id. For instance, the first time an application refers to a tuple, it can use its relation name and primary key. After the first read, the system returns the memory address of the tuple, and it is used for subsequent accesses. This avoids costly translations, but commits the system to leave the object in place, at least until the transaction that knows about the memory location terminates.

A second optimization is to eliminate the private buffer and to give transactions direct access to the object. The performance gains can be significant: if a transaction is simple, most of its time may be spent copying bits from and to buffers. By cutting this out, the number of instructions a transaction must execute can be cut in half or more. However, there are two potential problems now: once transactions can access the database directly, they can read or modify unauthorized parts; and the system has no way of knowing what has been modified, so it cannot log the changes. The best solution is to only run transactions that were compiled by a special database system compiler. For each database object access, this compiler will emit a code that checks for proper authorization, and logs every object modification [8].

I. Data Clustering and Migration

In a DRDB, data objects (e.g., tuples, fields) that are accessed together are frequently stored together, or clustered. For instance, if queries often look at a "department" and all the "employees" that work in it, then the employee records can be stored in the same disk page as the department they work in. In a MMDB there is, of course, no need to cluster objects. As a matter of fact, the components of an object may be dispersed in memory, as suggested in Sections II-D and II-E (e.g., tuples only have pointers to the data values stored elsewhere).

This introduces a problem that does not arise in conventional systems: when an object is to migrate to disk (or "vacuum cleaned" [25]), how and where should it be stored? There are a variety of solutions for this, ranging from ones where the users specify how objects are to be clustered if they migrate, to ones where the system determines the access patterns and clusters automatically. Our main point here is that migration and dynamic clustering are components of a MMDB that have no counterpart in conventional database systems.

III. Systems

Several database management systems for memory resident data have been proposed or implemented. These efforts range from pencil-and-paper designs (MM-DBMS, MARS, HALO) to prototype or testbed implementations (OBE, TPK, System M) to commercial systems (Fast Path). In the following, we describe some of these systems. Our descriptions are necessarily brief due to limited space. Furthermore, we focus our discussion on how these systems address the issues raised by memory resident data, which we discussed in the previous sections. This is summarized in Table I. (Blank entries in Table I mean that that particular aspect is not studied or implemented in the system, or that it is not described in the literature we have.) More detailed descriptions of these systems can be found in the references. Also, please note that our list is not comprehensive; we are simply discussing some representative systems.

TABLE I
SUMMARY OF MAIN MEMORY SYSTEMS

	Concurrency	Committ Processing	Data Representation	Access Methods	Query Processing	Recovery
MM-DBMS	two-pahse locking of relations	stable log tail by segment	self-contained segments, heap per segment, extensive pointer use	hashing, T-trees, pointers to values	merge, nested-loop, joins	segments recovered on demand, recovery processor
MARS	two-phase locking of relations	stable shadow memory, log tail				recovery processor, fuzzy checkpoints
HALO		in hardware, nearly transparent				physical, word-level log
OBE			extensive use of pointers	inverted indexes	nested loop-join, on-the-fly-index creation, optimization focuses on processor costs	
TPK	serial transaction execution	group committ, precommitt	arrays			two memory resident databases, fuzzy checkpoints
System M	two-phase locking, minimize concurrency	several alternatives	self-contained segments, heap per segment			various checkpointing, logging options
Fast Path	VERIFY/CHANGE for hot spots	group committ				

A. OBE

A main memory database manager has been implemented in conjunction with IBM's Office-By-Example (OBE) database project [1], [2], [26]. The system is designed to run on the IBM 370 architecture. Its focus is on handling *ad hoc* queries rather than high update loads.

Data representation in the OBE system makes heavy use of pointers. Relations are stored as linked lists of tuples, which in turn are arrays of pointers to attribute values. Indexes are implemented as arrays of tuple pointers sorted according to the attribute values on which the index was being built (i.e., inverted indexes).

Joins are computed using a nested-loop technique since sorting (for sort-merge joins) is not beneficial when the data are memory resident. An index may be created "on-the-fly" to compute a join.

Query processing and optimization focus on reducing processing costs since queries do not involve disk operations. Several important processing activities, such as creation of indexes and evaluation of query predicates, are identified and included in the cost formulas used by the query processor. Optimization techniques are also geared toward reducing or eliminating processor intensive activities.

B. MM-DBMS

The MM-DBMS system was designed at the University of Wisconsin [15], [17]. Like OBE, MM-DBMS implements a relational data model and makes extensive use of pointers for data representation and access methods. Variable length attribute values are represented by pointers into a heap, and temporary relations are implemented using pointers to tuples in the relations from which they were derived. Index structures point directly to the indexed tuples, and do not store data values. A variant of linear hashing is used to index unordered data, and T-Trees (a type of balanced binary tree with multiple values "stored" at each node) are used to access ordered data.

For recovery purposes, memory is divided into large self-contained blocks. These blocks are the units of transfer to and from the backup disk resident database copy. Commit processing is performed with the aid of some stable memory for log records and with a separate recovery processor. The recovery processor groups log records according to which blocks they reference so that blocks can be recovered independently after a failure. Blocks are checkpointed by the recovery processor when they have received a sufficient number of updates. A lock is set during the checkpoint operation to ensure that each block is in a transaction consistent state on the disk. After a failure, blocks are brought back into memory on demand and are brought back up to date using their log record groups.

MM-DBMS uses two-phase locking for concurrency control. Large lock granules (entire relations) are used.

C. IMS/VS Fast Path

IMS/VS Fast Path is a commercial database product from IBM which supports memory resident data [9]. Disk resident data are supported as well. Each database is classified statically as either memory or disk resident.

Fast Path performs updates to memory resident data at commit time. Transactions are group committed to support high throughput. The servicing of lock requests is highly optimized to minimize the cost of concurrency control. Record-granule locks are used.

Fast Path is designed to handle very frequently accessed data, since it is particularly beneficial to place such data in memory. It supports VERIFY/CHANGE operations for frequently updated objects. The VERIFY operation may be performed early in a transaction's lifetime to check the value of an object, but no locks are set for this operation. If the value

is judged acceptable, the actual update (along with another check of the value) is performed at commit time using a very short duration lock.

D. MARS

The MARS MMDB was designed at Southern Methodist University [6], [7], [12]. It uses a pair of processors to provide rapid transaction execution against memory resident data.

The MARS system includes a database processor and a recovery processor, each of which can access a volatile main memory containing the database. A nonvolatile memory is also available to both processors. The recovery processor has access to disks for the log and for a backup copy of the database.

The database processor is responsible for the execution of transactions up to the commit point. Updates do not modify the primary database copy until the updating transaction commits. Instead, the database processor records the update in the nonvolatile memory. If the updating transaction is aborted, the recorded update is simply discarded.

To commit a transaction, the recovery processor copies its update records into the database from the nonvolatile memory. The records are also copied into a nonvolatile log buffer. The recovery processor is responsible for flushing full log buffers to the log disks. Periodic checkpointing is also performed by the recovery processor. Checkpoints are fuzzy dumps of modified portions of the volatile database to the backup.

Concurrency is controlled using two-phase locking with large lock granules (entire relations).

E. HALO

Several main memory system designs, including MM-DBMS and MARS, propose dedicated processors for recovery related activities such as logging and checkpointing. HArdware LOgging (HALO) is a proposed special-purpose device for transaction logging [8]. It transparently off-loads logging activity from the processor(s) that executes transactions.

HALO intercepts communications between a processor and memory controllers to produce a word-level log of all memory updates. Each time a write request is intercepted, HALO creates a log entry consisting of the location of the update and the new and old values at that address. (HALO obtains the old value by issuing a read request to the memory controller.) These entries are maintained in nonvolatile buffers which are flushed to the disk when full. After buffering a log entry, HALO forwards the write request to the appropriate memory controller.

HALO also accepts several special commands from the processor. These commands (to begin, end, and switch transactions) are used to inform HALO of the identifier of the transaction that is currently being executed. HALO includes a transaction identifier with each log record. The processor can also cause a transaction's effects to be undone by issuing an abort transaction command to HALO.

F. TPK

TPK is a prototype multiprocessor main-memory transaction processing system implemented at Princeton University [18]. It runs on Firefly multiprocessors. TPK's emphasis is on rapid execution of debit/credit type transactions. It supports a simple data model consisting of records with unique identifiers. Transactions may read and update records using the identifiers.

The TPK system consists of a set of concurrent threads of four types: input, execution, output, and checkpoint. Input threads handle transaction requests and feed work to the execution thread via a queue, while output threads externalize transaction results and also make them available to the checkpointer. The execution thread executes the transaction code and is responsible for logging, and the checkpoint thread updates the stable backup database copy. Normally, TPK consists of a single execution thread and a single checkpointer, and one or more input and output threads. The execution thread executes transactions serially, thereby eliminating the need for transaction concurrency controls.

Two copies of the database (primary and secondary) are retained in-memory. The primary copy supports all transaction reads and updates. Copies of all updated records are placed (by the execution thread) into the log. TPK implements group commit to reduce the number of log disk writes per transaction.

The execution thread also places copies of each log record into a queue for the checkpoint process. The checkpointer reads these records and uses them to update the secondary in-memory database. Periodically, the secondary database is copied to the disk to complete a checkpoint. The purpose of the secondary database is to eliminate data contention between the checkpoint and execution threads during the checkpoint operation.

G. System M

System M is a transaction processing testbed system developed at Princeton for main memory databases [24]. Like the TPK prototype, System M is designed for a transactional workload rather than *ad hoc* database queries. It supports a simple record-oriented data model.

System M is implemented as a collection of cooperating servers (threads) on the Mach operating system. Message servers accept transaction requests and return results to clients. Transaction servers execute requested transactions, modifying the database and generating log data. Log servers move in-memory log data to disk, and checkpoint servers keep the disk-resident backup database up to date.

Unlike TPK, System M is capable of processing transactions concurrently. However, it attempts to keep the number of active transactions small. Two-phase locking is used for concurrency control. Both precommit and group commit are implemented for efficient log processing.

As in MM-DBMS, the primary database copy is divided into self-contained fixed-size segments, which are the units of transfer to and from the backup disks. Records are contained within a segment. Variable length fields are implemented using pointers into a per segment heap. Record index structures reside outside of the segments since they are not included in the backup database (nor are changes to indexes logged). Indexes are recreated from scratch after a failure, once the database has been restored from the backup copy and the log.

Since the focus of System M is empirical comparison of recovery techniques, a variety of checkpointing and logging techniques are implemented. System M can perform both fuzzy and consistent checkpoints (using a variety of algorithms) and both physical and logical logging. The physical organization of the backup database copy can also be controlled.

IV. Conclusion

In [10], it is argued that data that are referenced every 5 min or more should be memory resident (assuming 1K disk blocks). The 5-min number is arrived at by analyzing the dollar cost of accessing data in memory versus disk. The important thing to note is that as the price of a byte of main memory drops relative to the cost of disk accesses per second, the resulting time grows. That is, we can expect the " 5-min rule" to be the 10-min rule in the future, and so on. Thus as memory becomes cheaper, it becomes cost effective to keep more and more data permanently in memory. This implies that memory resident database systems will become more common in the future, and hence, the mechanisms and optimizations we have discussed in this paper will become commonplace.

References

[1] A. C. Ammann, M. B. Hanrahan, and R. Krishnamurthy, "Design of a memory resident DBMS," in *Proc. IEEE COMPCOM Conf.*, 1985.

[2] D. Bitton, M. B. Hanrahan, and C. Turbyfill, "Performance of complex queries in main memory database systems," in *Proc. Int. Conf. on Data Engineering*, Feb. 1987, pp. 72–81.

[3] G. Copeland, R. Krishnamurty, and M. Smith, "The case for safe RAM," in *Proc. 15th Int. Conf. on Very Large Databases*, Amsterdam, 1989.

[4] G. Copeland, M. Franklin, and G. Weikum, "Uniform object management," in *Proc. Int. Conf. on Extending Database Technology*, Venice, Italy, Mar. 1990, pp. 253–268.

[5] D. J. DeWitt *et al.*, "Implementation techniques for main memory database systems," in *Proc. ACM SIGMOD Conf.*, June, 1984.

[6] M. H. Eich, "A classification and comparison of main memory database recovery techniques," in *Proc. Int. Con. on Data Engineering*, Feb. 1987, pp. 332–339.

[7] ——, "MARS: The design of a main memory database machine," in *Proc. Int. Workshop on Database Machines*, Oct. 1987.

[8] H. Garcia-Molina and K. Salem, "High performance transaction processing with memory resident data," in *Proc. Int. Workshop on High Performance Transaction Systems*, Paris, Dec. 1987.

[9] D. Gawlick and D. Kinkade, "Varieties of concurrency control in IMS/VS Fast Path," *Data Eng. Bull.*, vol. 8, no. 2, pp. 3–10, June 1985.

[10] J. Gray and F. Putzolu, "The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time," in *Proc. 1987 ACM SIGMOD Conf.*, San Francisco, CA, May 1987, pp. 395–398.

[11] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann, 1992.

[12] L. Gruenwald and M. H. Eich, "MMDB reload algorithms," in *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, pp. 397–405.

[13] R. B. Hagmann, "A crash recovery scheme for a memory-resident database system," *IEEE Trans. Comput.*, vol. C-35, pp. 839–842, Sept. 1986.

[14] M. Y. Kim, "Synchronized disk interleaving," *IEEE Trans. Comput.*, vol. C-35, pp. 978–988, Nov. 1986.

[15] T. J. Lehman and M. J. Carey, "Query processing in main memory database management systems," in *Proc. ACM SIGMOD Conf.*, Washington, DC, May, 1986.

[16] ——, "A study of index structures for main memory database management systems," in *Proc. 12th Conf. on Very Large Data Bases*, Aug. 1986.

[17] ——, "A recovery algorithm for a high-performance memory-resident database system," in *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, pp. 104–117.

[18] K. Li and J. F. Naughton, "Multiprocessor main memory transaction processing," in *Proc. Int. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, Dec. 1988, pp. 177–189.

[19] D. Patterson *et al.*, "RAID: Redundant arrays of inexpensive disks," in *Proc. ACM SIGMOD Conf.*, Chicago, June 1988.

[20] P. Pucheral, J.-M. Thevenin, and P. Valduriez, "Efficient main memory data management using the DBGraph storage model," in *Proc. 16th Conf. on Very Large Data Bases* Brisbane, 1990, pp. 683–695.

[21] A. Reuter, "Performance analysis of recovery techniques," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 526–559, Dec. 1984.

[22] K. Salem and H. Garcia-Molina, "Disk striping," in *Proc. Int. Conf. on Data Engineering*, Los Angeles, CA, Feb. 1986, pp. 336–342.

[23] ——, "Checkpointing memory-resident databases," in *Proc. Int. Conf. on Data Engineering*, Los Angeles, CA, Feb. 1989, pp. 452–462.

[24] ——, "System M: A transaction processing testbed for memory resident data," *IEEE Trans. Knowl. Data Eng.*, vol. 2, pp. 161–172, Mar. 1990.

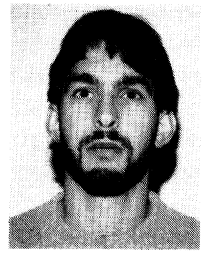
[25] M. Stonebraker, "Managing persistent objects in a multi-level store," in *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, pp. 2–11.

[26] K.-Y. Whang and R. Krishnamurthy, "Query optimization in a memory-resident domain relational calculus system," *ACM Trans. Database Syst.*, vol. 15, no. 1, pp. 67–95, Mar. 1990.

Hector Garcia-Molina (S'70–M'79) received the B.S. degree in electrical engineering from the Instituto Tecnologico de Monterrey, Mexico, in 1974, and the M.S. degree in electrical engineering and the Ph.D. degree in computer science from Stanford University, in 1975 and 1979, respectively.

He is currently a Professor with the Department of Computer Science, Stanford University, Stanford, CA. From 1979 to 1991 he was on the faculty of the Computer Science Department, Princeton University, Princeton, NJ. During 1991, he was also the Associate Director of the Matsushita Information Technology Laboratory in Princeton. His research interests include distributed computing systems and database systems.

Dr. Garcia-Molina is a member of the ACM.



Kenneth Salem (S'83–M'88) received the B.S. degree in electrical engineering and applied mathematics from Carnegie-Mellon University in 1983, and the Ph.D. degree in computer science from Princeton University in 1989.

He is currently an Assistant Professor with the Department of Computer Science, University of Maryland, College Park, and a Staff Scientist with NASA's Center of Excellence in Space Data and Information Sciences, located at the Goddard Space Flight Center. His research interests include database and operating systems and transaction processing.

Dr. Salem is a member of the ACM.