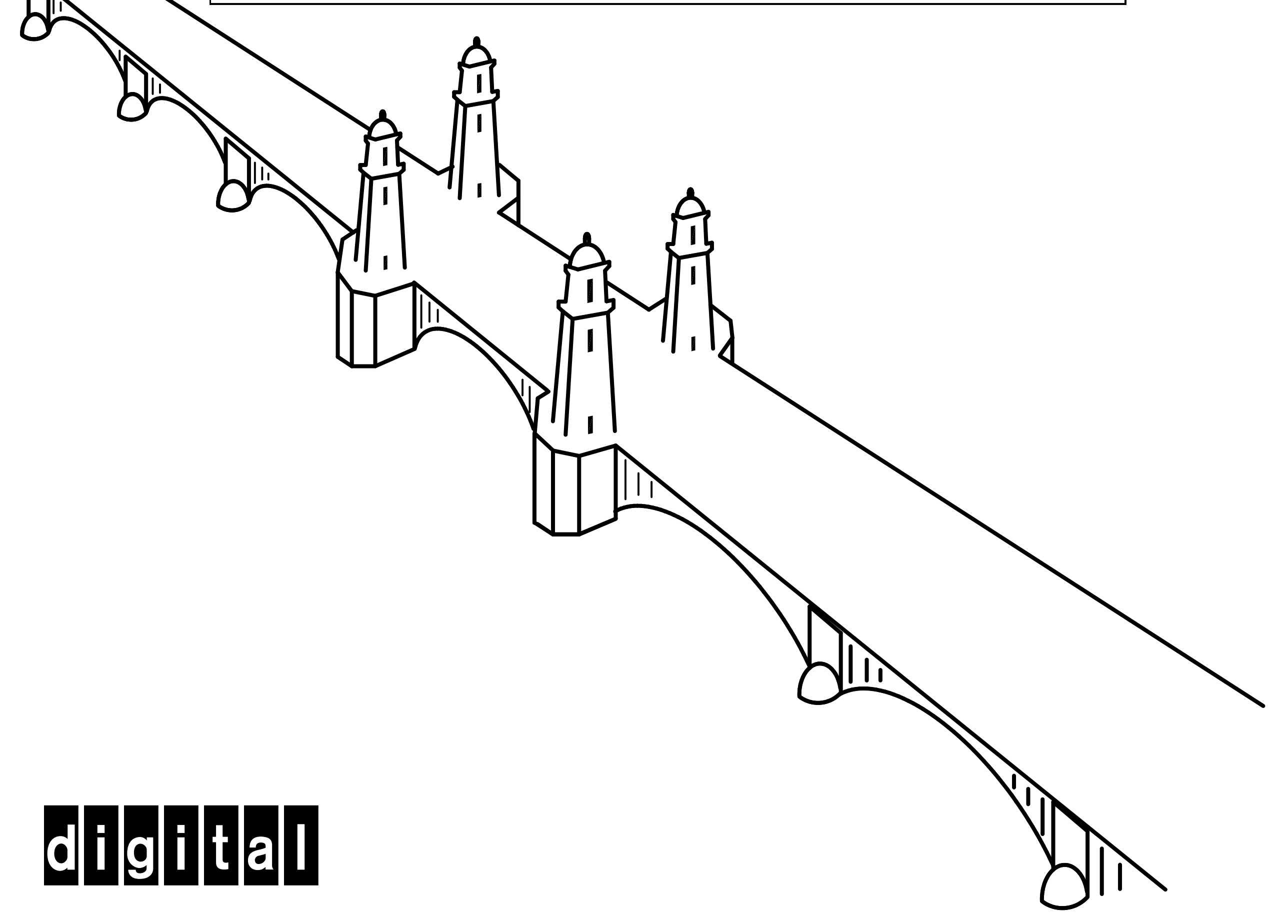
How the Rdb/VMS Data Sharing System Became Fast

David Lomet Rick Anderson[1]
T. K. Rengarajan[1] Peter Spiro

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/4 May 29, 1992



CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

digital™

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

How the Rdb/VMS Data Sharing System Became Fast

David Lomet   Rick Anderson[1]
T. K. Rengarajan[1]   Peter Spiro [1]

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/4   May 29, 1992

Abstract

Recent versions of Rdb/VMS have shown dramatic performance increases compared with earlier versions. Performance enhancements have culminated in an 80% improvement between Rdb/VMS V3.0 and Rdb/VMS V4.1 when executing on the same hardware. This has vaulted Rdb/VMS to an industry leadership position in $/TPS and to a very competitive position in peak TPS. While code paths have been shortened, the primary means of achieving this performance gain has been through reducing I/O accesses and distributed locks. This paper outlines how this was done.

Keywords: database performance, locking, buffering, recovery, commit processing

[1]Digital Equipment Corp., Database Systems Group, Nashua, NH

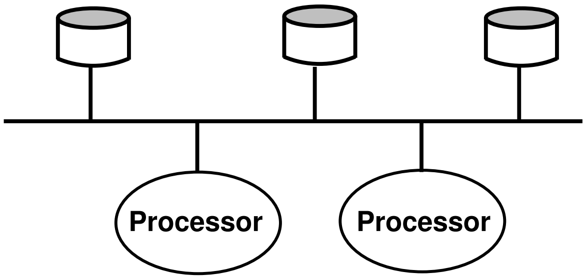


Figure 1: A shared disk system that supports a data sharing database system.

1 Introduction

1.1 Data Sharing Systems

A data sharing database system is one in which multiple servers manipulate data on disks that all the servers can access simultaneously. Rdb/VMS [1] is unusual among database systems in being a data sharing system. It is designed to execute on a VAXcluster [4], a system configuration called a "shared disk" system because multiple processors can access a common set of disks. Figure 1 schematically portrays a shared disk system configuration that enables data sharing.

Data sharing systems have an interesting blend of advantages and difficulties. The difficulties include the need to use a distributed lock manager (DLM) [7] in order to mediate requests for the common resources. The DLM is usually used to provide cache coherency as well, where the problem is to ensure that the multiple servers see a consistent view of the database. Additional problems involve providing recovery when some but not all servers for some data fail, a situation that can arise with data sharing but that is impossible with partitioned systems.

However, data sharing systems provide some substantial advantages.

- Unlike partitioned systems, Rdb/VMS does not need to be carefully

[1] Rdb/VMS, VMS, VAX, and VAXcluster are trademarks of the Digital Equipment Corporation.

administered so as to balance the work load across multiple servers. (Careful placement of data on disks remains a problem.)

- Users can scale their applications easily by adding new processors or new disks and controllers. Rdb/VMS flexibly adjusts to the new environment.

- Availability is enhanced because servers on any processor can all access the data. Thus, so long as some VAX in a VAXcluster is in operation, the database is available.

The above features are highly desired by users executing production applications, and Rdb/VMS has long made support for such applications a high priority [3].

1.2 The Changing Hardware Playing Field

The revolutionary advances in processor performance, coupled with the ever larger quantities of storage available, have reset user expectations as to the performance of database systems. Users expect continuous improvement in the price and performance of the database processing that they are currently doing. They also expect that new applications that were previously infeasible, should now become a staple part of their application suite.

Database engineers know that meeting user expectations requires much more than taking a free ride on hardware improvements. While processor speeds have improved enormously, and storage costs have plummeted, the performance balance in today's systems has changed dramatically. The key point of this is the following:

POINT: The performance of any system is limited to that which can be provided by its most performance constrained component. This component then becomes the system **bottleneck**.

Today's systems have less I/O bandwidth per processor MIPS capability than their predecessors. Even more dramatically, today's systems fall short on their ability to provide I/O accesses/second per MIPS. Consider the change in disk and processor characteristics as shown in Table 1 below for the years 1980 and 1990. (The numbers have been rounded for easy comparison.)

Single Disk Accesses vs Single Chip Processor Performance			
Year	Accesses/Sec	MIPS	(Accesses/sec)/MIPS
1980	36	4	9
1990	55	27.5	2

Table 1: Changing I/O to Processor Performance

Thus, in 1980, a one chip, one disk system could support no more than one I/O access for every 111K instructions. In 1990, such a system could support no more than one I/O access per 500K instructions. The result of executing a database system in 1990 that was designed for 1980 system components is a bottleneck on I/O. This can mean processors that are less than fully utilized, perhaps dramatically so, as they wait for I/O to complete.

1.3 Scaling to High Performance

Customers seem to need ever larger databases serving an ever larger user community. The scaling problem for data sharing systems is that code path tends to increase as more servers are added to the system. This is true whether the servers are within a single symmetric multiprocessor (SMP) node or whether they are instantiated on additional nodes in a VAXcluster. In both cases, there is an increase in the number of remote lock requests that may have to be serviced because each server becomes a smaller part of the overall system. Transferring active data from one server to another can increase as well, which results in added I/O activity. It is important to prune this locking and I/O activity so as to contain the problem.

1.4 Adjusting to The New Requirements

When planning for these new requirements, it is not sufficient to simply try to make everything more efficient. In particular, simply shortening code path may have, as its only effect, that the processor utilization declines and overall performance is unchanged. This is the same effect that arises when a faster processor replaces a slower one while the system is bottlenecked elsewhere.

What is needed is to direct attention to the system bottlenecks. That is, it is necessary to reduce I/O accesses for the same functionality. In a data sharing system, reducing messages involves reducing the number of distributed DLM locks that are needed. This clearly has a direct impact on user perceptions of response time as there are fewer I/O and message waits. Perhaps less expected is that the result of this work is shorter code paths as well!

Concentration on I/O and messages results in shorter execution paths. This is a consequence of avoiding the substantial processor instruction execution costs of I/O and message handling. Typically, I/O and message instruction costs are multiple thousands of instructions. These instruction costs can be a substantial fraction of the path length of a TPC-A transaction.

What is unique about much of the work described in the following sections of this paper is that it was done in the context of a process-based data sharing system. This work makes Rdb/VMS truly unique in that the advantages of data sharing are preserved while providing world class performance and price performance.

1.5 Paper Organization

In the remaining sections of this paper, the major innovations that we exploited in order to give Rdb/VMS performance leadership are described. There are four areas that were attacked.

Lock Handling: The number of locks that need to be acquired per transaction were reduced by techniques that involve having each lock cover more resources (when that is possible) coupled with having locks that span several transactions.

Global Buffers: Rdb/VMS executes in each user's process, which provides a very responsive interface to the database. A problem is that each user had his own buffer of database pages. Pages shared by several user processes then needed to be switched back and forth between processes. Providing node wide (global) buffers greatly reduces this switching.

Recovery: A database system must ensure that a committed transaction is durable and that uncommitted transactions that fail can be erased. The goal here was to minimize the I/O accesses needed to store data

Acronym	Definition
AIJ	After-Image Journal (for redo)
ALG	Adjustable Locking Granularity
AST	Asynchronous System Trap
DBR	Database Recovery Process
DLM	Distributed Lock Manager (of VMS)
PSN	Page Sequence Number
RUJ	Run-Unit Journal (for undo)
SMP	Symmetric Multi-Processor

Table 2: Glossary of Acronyms

stably. The trick was to reduce the amount of data that needs to be stably stored, and to exploit mainly sequential I/O for accomplishing this. Which server recovers which part of the database needed also to be solved.

Commit Processing: The sequence of steps needed in order to commit transactions has required the acquisition of multiple DLM locks and the scheduling of multiple writes. How both locks and I/Os were reduced is described. Of particular importance is the reduction in the length of time that locks are held.

Rdb/VMS, like most complex systems, exploits acronyms to describe concepts, components, and features. These are used to shorten the description of how it works. Table 2 below contains a summary of all acronyms introduced in the description of the subsequent sections.

2 Lock Handling

2.1 The Heart of Data Sharing

Rdb/VMS is a data sharing system, in large part, because it exploits the VMS distributed lock manager (DLM) for concurrency control. The DLM provides services for naming and locking cluster-wide resources and for performing cluster-wide synchronization. DLM is a very efficient lock manager when lock requests are handled within a single node of a cluster. However, lock requests across nodes require that messages be sent, making the distributed functionality expensive to use.

Rdb has always taken measures to minimize its use of distributed locks [2]. In particular, it exploits lock de-escalation for this. That is, a process will acquire a lock on a large granule (e.g. file) so as to permit it to operate on pages and records of the file without needing to make additional lock requests to the DLM.

Should another process want to access some of the data of a file, the process holding the file lock is notified. This holding process can de-escalate its file lock to tuple or page locks. This permits other processes to acquire locks on tuples or pages not of current interest to the first process. Hence, only if a conflict actually occurs is locking at page or tuple granularity performed via the DLM. Multi-granularity locking and intention locks are used to realize this. These lock modes are supported by the DLM.

DLM lock owners are processes, not transactions. These owning processes provide interrupt handlers that respond to DLM interactions outside of the main thread of the process. The interrupt handler within a process is called an AST (asynchronous system trap) routine. It responds to the DLM, e.g., by deciding on a response to DLM reported lock conflicts. The fact that locking is process based can be viewed as a limitation, but it is also possible to exploit this DLM characteristic. This is described below.

2.2 Lock Carry-over

2.2.1 Locking Locality in a Data Sharing System

If there are many processes competing for the same lock or small set of locks, then lock thrashing will rapidly set in. The result is that very substantial overhead will be incurred responding to lock conflicts within AST routines. Rdb takes pains to minimize this thrashing, as described above. It is also possible for Rdb/VMS users to reduce the level of lock contention.

The most common circumstance is that users are simply accessing different resources, e.g. records(tuples) or files(tables). Most of the data in a database is cold data that is rarely accessed. Further, when it is accessed, there is usually some locality to the reference pattern. For example, a user who completes one ATM transaction subsequently initiates a second transaction against the same account.

Even for data that is hot, conflicting lock accesses can be avoided. It is commonplace for debit/credit style transaction execution to be mediated by a transaction monitor, e.g. ACMS. The transaction monitor routes the transaction to a server that is dedicated to handling some part of the database. This is called partitioning. Partitioning is very useful for improved performance. For example, it helps ensure that a database server has the hot resources in its cache. For distributed locking, such partitioning results in a single process handling all requests against some small set of hot resources.

The end result for both cold data and partitioned hot data is that it is unlikely that another process will access this data concurrently. This makes locks on the data low-conflict locks.

2.2.2 Exploiting Locality and Low Conflict

Prior versions of Rdb/VMS did traditional database locking for logical resources (records and "logical areas"), i.e., database locks were acquired during a transaction and held, as prescribed by the strict two phase locking protocol (strict 2PL) until the transaction was completed. Despite the fact that processes own locks when using the DLM, a process relinquished its logical locks at end of transaction. Should the next transaction executed by a process have wanted to lock the same resources, these locks had to be re-acquired, as if the process had never seen the locks before.

Another strategy is possible. Instead of acquiring and releasing locks during every transaction, locks acquired in prior transactions can be carried over to the present transaction. We call these locks carry-over locks, and this optimization is called the carry-over optimization.

2.3 The Carry-Over Realization

2.3.1 Multi-granularity Locking and Lock Conflicts

If transactions access several files, etc., a process executing those transactions can end up holding a large number of locks. Further, in order to respond appropriately to conflicting lock requests, it is necessary to distinguish locks actually in use (by the current transaction) from carry-over locks that are not used currently. The later category can be released on receipt of a conflict message from the DLM. The former need to be retained until end-of-transaction. We refer to these former locks as IN-USE locks and mark them appropriately.

The carry-over optimization is applied at the root granule of a tree denoting the multi-granularity locking hierarchy, called the ALG (adjustable locking granularity tree) root. A conflicting request at the ALG root causes the lock on the root to be de-escalated. That is, explicit locks are posted for locks below it in the tree that guard resources being used by the current transaction. Then, the strong mode lock on the ALG is replaced with a weaker "intention" mode lock.

Which locks are carried over from one transaction to the next depends on whether the ALG root lock is strong or weak. If strong, it indicates that no conflicting requests for resources in its resource tree have been received. If the ALG root lock is weak, that indicates that it has been de-escalated to satisfy a request for one or more of the resources in the tree. Thus, lock trees with strong ALG root locks have the locks carried over. Those with weak ALG root locks are all released.

2.3.2 Interactive *NOWAIT* Transactions

Rdb supports a feature called *NOWAIT* transactions to enable interactive users to avoid waiting for locks. That is, should a locking request of the interactive user threaten to block, because another transaction has the lock in a conflicting mode, a *NOWAIT* transaction will not hang, but rather returns immediately to the user with a lock conflict message. This permits interactive users to perform other work, and then perhaps subsequently re-attempt the previously denied request.

Carry-over locking creates a potential problem for interactive users with *NOWAIT* transactions. Carry-over locks may permanently deny such users access to the data they are guarding, even though there are multiple occasions in which the data is actually available outside of conflicting transactions.

To preclude permanent exclusion for *NOWAIT* transactions, a *NOWAIT* transaction broadcasts its presence to all processes of the data sharing system. This is done via the capability of the DLM to support global locks and to resolve conflicts on the global locks via notification messages sent to ASTs. Upon notification, all other processes disable carry-over locking until such time as there are no more *NOWAIT* transactions executing.

—– System Configurations —–		
One Node	Two Nodes	Three Nodes
16%	61%	67%

Figure 2: Improvement in Debit/Credit TPS performance by carry-over locking.

2.4 Lock Reduction Achieved

An important attribute of our implementation is that applications that are not partitioned are not adversely impacted by carry-over locking, as such applications implicitly disable carry-over locks. And for applications for which accesses are partitioned, the performance gain can be substantial. The TPC-A (debit/credit) benchmark is one such application.

The TPC-A benchmark was executed both with carry-over locking enabled, and with it disabled. The system configuration varied from all processes executing on a single node, to being on two, then three nodes. In Figure 2, we present the percent improvement achieved by the carry-over optimization for each configuration. The reduction in locks requested was over 50%.

3 Global Buffers

3.1 Process-Private Local Buffers

Prior to V4.1 of Rdb/VMS, all processes accessing the database managed database pages in process-private buffer pools. This is illustrated in Figure 3a. Cache coherency between buffer pools was achieved by the use of the DLM. Essentially, whenever a process, even on the same node, wished to access a database page held by another process, it needed to

1. read the page from disk, thus wasting disk I/O bandwidth;

2. maintain a separate copy of the page in its private buffer, thus wasting physical memory.

Database buffers can hold a number of database pages, which in turn can be a number of disk blocks. For the purposes of our discussion, we assume one page per database buffer. To read or update a page, a user gets a DLM page lock. The DLM maintains version numbers associated with each lock. These version numbers are contained within what is called the "lock value block". The current version number of the page in the lock value block is compared with the old version number of the page in the buffer pool to decide if the page should be read from disk.

3.2 Multi-process Global Buffers

The global buffers feature implemented in Rdb V4.1 overcomes these disadvantages. With global buffers, there is only one pool of buffers for a database for each node, which is kept in a global section. The global section is mapped into a portion of each user process's virtual memory. Only one copy of any database page is maintained in the global buffer. Additionally, only one process reads the page from disk. Other processes simply reference the already present page in the global buffer.

The management of global buffers is performed in a distributed fashion by cooperating database processes. There is no special process that manages the global buffer pool. Buffers are managed via a two-level scheme. This scheme is illustrated in Figure 3b.

Level 1: Allocate Sets: Every process maintains an "allocate set" of global buffers. The allocate set is the collection of buffers that previously had been the process private buffer pool. The allocate set is locked by the process in the global buffer pool (see below).

A process performs intra-node synchronization using local DLM locks to bring a buffer into its allocate set. When many processes read the same buffer, it can be in the allocate sets of all of them. Once a buffer is brought into the allocate set of a process, no other process can remove the buffer from the global buffer pool.

The maximum size of an allocate set is statically determined by each process at bind time. Once an allocate set has grown to the maximum size, it is necessary to purge buffers from the allocate set before other buffers can be brought in. The victim buffer to be purged from the allocate set is determined by an LRU algorithm.

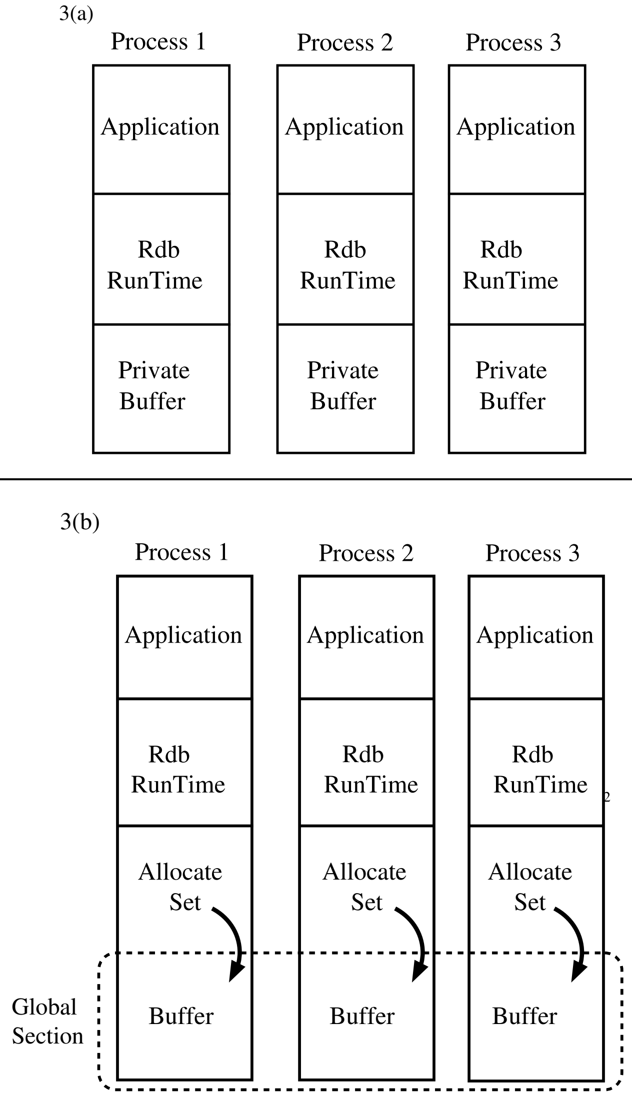


Figure 3: (a) Process mapping for private buffers. (b) Process mapping for the global buffer feature.

Level 2: Global Buffer Pool:

The "global buffer pool" is the global storage pool that contains all the allocate sets of all the processes on a node. Searching for a database page in the global buffer pool is made fast by the use of a hash table. The hash table is an array of queue headers for queues of buffers that have hashed to the same slot in the table. Each queue element stores a count indicating in how many allocate sets the associated global buffer is included.

Searching for victim buffers to be replaced by new buffers is determined by a pseudo-LRU queue of possibly-free buffers. A buffer that is not in the allocate set of any user is enqueued. When a victim is needed, so as to provide a free buffer for new data, potential victim buffers are dequeued from this pseudo-LRU queue. The selected victim is the first buffer that is not in an allocate set. This queue contains only potential victims because a global buffer may become part of an allocate set after it has been enqueued.

This two-level buffer management has some interesting characteristics. Each process can request a different size allocate set when a session is opened with a database, based on the application. A process can access buffers in this allocate set without any need for synchronization or process context switches. It is very easy to lock a set of database buffers in the global buffer pool. There is minimal interference between the access patterns of different processes. For example, a process performing a sequential read cannot force out of cache a set of buffers that another process needs, e.g. for a nested-loops join.

The locking protocol for cache coherency of global buffers is very similar to that used for local buffers. A user still gets the same lock and checks versions. Global buffers may not go away with process failures. But page locks owned by a process are released by the DLM when a process fails. In order to preserve version coherence (via lock value blocks in a VAXcluster environment), system-owned locks are held (in NL mode) on all pages in the global buffer pool.

3.3 Achieving High Availability

As discussed above, operations like movement of buffers into and out of user allocate sets involves updates of global buffer data structures. A process failure during such operations can leave the global buffer data structures inconsistent and hence unusable. This problem is directly a result of the distributed buffer management by cooperating processes and can adversely affect system availability when the global buffer pool is accessed by a large number of processes.

To solve this problem, we need to do global buffer operations atomically. We achieve this by a mechanism called global buffer transactions. The set of global buffer data structures is considered to be a database and well-known db locking and logging strategies are employed. We lock the global buffer data structures (using node-private local DLM locks) before updating them. Then we log the before images to an in-memory log. This log never has to be flushed to disk, since its only purpose is to guarantee atomicity of operations with respect to one global buffer pool. In particular, it does not have to survive system failure.

If a process fails during the update, a recovery process (DBR) undoes the global buffer transaction. As part of recovery for a failed user, DBR also purges all buffers from the allocate set of the failed user. These can be considered as compensating transactions for previously executed global buffer transactions. Such recovery of global buffer transactions happens before the database transaction recovery is attempted.

3.4 Performance Impact

The use of global buffers helps most when data is shared between different users. The performance improvement depends on the extent of data sharing. Experiments with simple applications show performance improvements of about 25%, but the benefit can be much higher. The overhead of global buffers is in the extra synchronization. This is minimized by the two-level buffer scheme. When there is absolutely no sharing, experiments indicate a performance overhead of approximately 9% for the same simple applications. This is the worst case for the performance overhead of global buffers.

Note that global buffering provides a performance improvement in cases where carry-over locking cannot be used. These cases are when data is shared between users, potentially disabling the carry-over lock optimization. Thus, it provides its performance boost to applications that would not otherwise see an improvement. This is particularly important when the data sharing nature of Rdb/VMS is being exploited.

4 Recovery Techniques

4.1 Pure Undo Recovery

In previous releases, Rdb/VMS utilized an undo/no-redo logging technique [6]. This required that all modified database pages be flushed to disk before a transaction could commit. Because all modified pages were flushed to disk at commit time, Rdb/VMS never needed to redo the effects of a failed transaction. Thus, the previous releases favored fast recovery at the expense of longer commit processing.

If the commit sequence forces all updates to disk before the transaction is allowed to commit, then any abnormal termination can be recovered by performing only undo recovery. So this method is characterized by very fast recovery processing at the expense of less than optimal commit processing.

4.2 Redo Logging via AIJ

Rdb/VMS version 4.1 allows a database to utilize two different recovery strategies. The database can be configured to flush all updated database pages back to disk during commit as before; or the database can avoid flushing the database pages to disk and simply log after-image records to a journal. This second method is called redo logging.

When redo logging is enabled, at commit time a transaction's dirty pages are NOT flushed to the disk. Instead, commit information is submitted and flushed to the After Image Journal (AIJ). The transaction is then marked as committed. The benefit of this feature is that frequently accessed database pages do not need to be written to disk as often.

Even when using redo logging, the undo journal containing before images (RUJ) is still utilized. This means that if a marked, but uncommitted database page is flushed to disk before the end of the transaction (for instance, because of cache overflow), the before-image information will be flushed to the journal. However, transactions that do not flush uncommitted marked pages to disk will not have to incur the cost of the before image journal.

The two recovery methods not only differ in the work required during commit processing, but they also differ in the recovery processing which would be necessary in the event of an abnormal transaction termination.

With redo recovery, commit is very efficient: the only requirement is that the transaction's after-image records must be flushed to the AIJ log. However since committed updates for database pages have not been flushed to disk, a failure requires DBR to perform redo recovery in addition to any undo recovery that might be necessary. Hence redo recovery is characterized by very fast commit processing and expensive recovery processing. This is usually considered to be a desirable trade-off.

After a process completes a number of transactions, more and more updated, committed pages will exist in the process's buffer pool. If the process dies, the time needed to perform redo recovery is related to the size of the AIJ file, and the number of updates the recovery process must redo. For example, assume a database is processing 100 transactions per second with each transaction generating 1K of AIJ log data. Thus the database is generating almost a gigabyte of log data every three hours. If a user process had been running for three hours with redo recovery enabled, and then aborted abnormally, DBR would have to scan a gigabyte of log data in order to correctly redo all the transactions for the dead process. In most circumstances this would be an unacceptable.

4.3 Checkpoints For Bounding Redo Recovery

4.3.1 Checkpoints

In order to bound the recovery time, Rdb/VMS employs checkpoints which allow DBR to scan a smaller section of the AIJ log. In Rdb/VMS, each database user individually performs their own checkpoint. which will occur after the completion of a particular transaction. A checkpoint is a three step sequence: first the process flushes all its updated committed database pages back to disk; then the process submits and flushes a checkpoint record to the AIJ log; finally the process records, in the database root file, the location of this checkpoint record.

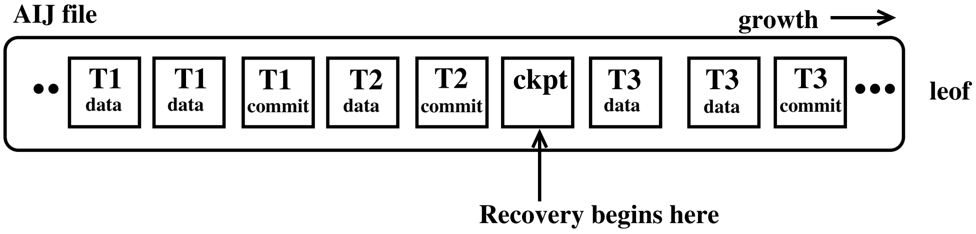


Figure 4: Recovery can ignore transaction T1 and transaction T2 since they were committed and flushed to disk before the checkpoint record. Transaction T3 must be redone.

By flushing all the updates to disk we ensure that the recovery process will not have to redo any of these updates. Furthermore by writing the checkpoint record to the AIJ file, a guaranteed safe starting point is identified for DBR. In other words, DBR never has to perform redo for any AIJ records that precede the checkpoint record. Their changes have been flushed and are durably recorded in the database pages on disk.

The recovery mechanism will be detailed further in the paper, however in its simplest description the recovery process has to scan the AIJ file from the checkpoint record to the end of file, possibly redoing transactions for the aborted process. Thus the time required for the recovery is directly related to size of the AIJ growth since the database user wrote its last checkpoint record to the AIJ file. This is illustrated in Figure 4 below.

4.3.2 Checkpoint methods

As mentioned above, Rdb/VMS does not perform a global checkpoint for all database users, instead each user completes its own checkpoint when a certain event dictates a checkpoint is desirable. Since applications can be so varied, Rdb/VMS has provided three different mechanisms for triggering a user checkpoint:

AIJ File Growth: The DBA can set the AIJ file growth checkpointing[4] parameter to be some number of blocks. A database user will flush a new checkpoint record whenever the AIJ file has grown at least this number of blocks since the user flushed its last checkpoint record.

Transaction count: The transaction count method forces a database user to checkpoint after it has completed certain number of transactions. This setting is often useful when the behavior of the application is very well understood.

Time: The time method utilizes the elapsed clock time since the last checkpoint. This parameter is often useful as a catch-all to prevent users from running for a very long time between checkpoints.

The method which is most deterministic in bounding recovery time is the AIJ file growth parameter. The duration of recovery is reasonably well known: the time necessary to scan some set number of blocks of data and possibly redo some database updates.

4.4 Logging Requirements for Recovery

Because users checkpoint independently and may abort independently, recovery has to proceed in a slightly different manner from many other redo recovery schemes. In Rdb/VMS the recovery process must be able to recover for an individual user. To solve this problem each AIJ record submitted to the AIJ file contains an unique ID which identifies a different database user.

Another factor is that one user's AIJ records may be 'outdated' by another user's subsequent data modification and respective AIJ record. For example, assume user U1 updates record R1 and commits. Then user U2 updates record R1 and also commits. Then user U1 aborts. At this point the AIJ file is in the state depicted by Figure 5.

Since the recovery process is only performing redo for U1, it must be selective about which updates it must redo. If an AIJ record is made obsolete by a subsequent AIJ record, the recovery process must not redo the change as it will produce in an incorrect result. To solve this problem, Rdb/VMS tags each AIJ record with a page sequence number (PSN) [5].

A PSN exists on all live data pages. Whenever an AIJ record is submitted, it includes the PSN of the page before the transaction made the data modification. Then the transaction increments the PSN. So the PSN is an identifier of the state of the page when the modification occurred. In addition, it is important to state that whenever a page migrates from one database user to another, it must first be flushed to disk. This guarantees that at most one user has unposted updates for a page.

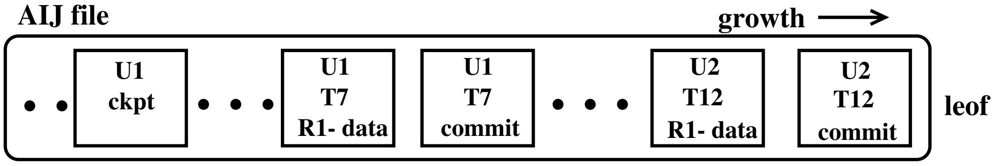


Figure 5: Record R1 was updated by U1 in transaction T7. Subsequently, R1 was updated by U2 in transaction T12. If U1 aborts, the recovery process must not redo the update to R1 done by T7 because it would reverse the effects of T12.

When DBR performs redo, it only redoes an AIJ record, if the PSN in the AIJ record matches the PSN it finds on the page it just read off desk. This is a key point. In other words, if the PSN matches, it means the update (reflected in the AIJ file) did not get propagated to disk and consequently the operation must be redone. If the PSN does not match, it means the update must have been flushed to disk so DBR does not redo the operation.

4.5 Redo Recovery

When a transaction aborts abnormally, Rdb/VMS automatically starts up a recovery process. First the recovery process reads the database root file to determine the VBN in the AIJ file of the aborted user's checkpoint record. The recovery process now has to scan the AIJ file from the checkpoint VBN to the end-of-file, searching for committed AIJ records submitted by the aborted user.

When the recovery process finds a candidate AIJ record to redo, it then checks the PSN from the AIJ record, with the PSN on the database page in the database. If the PSN from the AIJ record does not match the data page, the recovery process can avoid the redo. This will occur if another database user had subsequently modified the page after the aborted user had submitted its AIJ record. In other words, the update had been flushed to disk. However if the PSNs match, the modification never got flushed to disk and the recovery process must redo the operation logged on the AIJ[5] record.

After the recovery process has scanned to the end of the AIJ file, the redo phase is completed, and the recovery process performs the undo phase of recovery.

In the event of a system crash, recovery immediately proceeds if the database is accessed on any existing node in a VAXcluster. In a single-node system, recovery would begin automatically after the system was rebooted and the database was accessed. In both scenarios, the actual recovery proceeds identically to the transaction recovery described above, that is, the recovery process scans the AIJ file and performs redo for each failed transaction.

5 Commit Process

5.1 Group Commit

5.1.1 Rationale

The purpose of group commit is to batch a series of related, expensive operations such that the cost of processing the serial portion of the operation, such as I/O, is amortized across many transactions. In the case of journaling the after-image records, or updating the transaction state information in the database root file, this also allows the system to fully utilize the disk device bandwidth. Version 4.0 of Rdb/VMS was the first version of Rdb/VMS to support group commit [8].

The Rdb/VMS transaction commit operation is actually performed in two distinct phases: an after-image journal posting phase and a transaction state information posting phase. Each of these phases is performed using a separate but similar grouping mechanism, which will be described below.

5.1.2 After-Image Journal Posting

As data records are modified, after-images of the modified records are stored in a per-process cache. When the transaction commits, or the cache overflows, the contents of the cache are formatted into variable-length records which are then appended to a "after-image posting queue".

When the process cache has been completely submitted to the queue, the process “sleeps” for a small amount of time. This processing pause allows other committing processes to flush their commit information as well to the same after-image posting queue. The purpose of this pause is to reduce the use of distributed locks. The expectation is that the locking protocol of the next paragraph will frequently be avoided.

When the process awakens from the sleep, it checks to see if its records in the queue have been processed. If the queue entries have been processed, the transaction commit operation proceeds to the next phase. Otherwise, the process attempts to acquire an exclusive lock on a special lock resource. If the lock is immediately granted, the successful process is known as the “group poster”. The blocked processes know that another process is performing the group post operation, and they simply wait until their queue entries have been processed.

The group poster process will format all after-image posting queue records into a large buffer and perform a single write operation to the after-image journal, appending the buffer to the current end-of-file. Once the I/O is complete, the posting queue entries are removed and the lock is released.

5.1.3 Transaction State Posting

After receiving notification that the group after-image posting operation has been completed, each transaction submits an entry to the ”transaction state” queue indicating that it has committed successfully. If the transaction submitted the first entry on the queue, that process is designated as the ”group poster”. Otherwise, the process ”sleeps” for a small amount of time; this processing pause allows other committing processes to also submit entries to the queue, so that the group poster can process multiple queue entries in a single pass.

The group poster process will update the database on-disk transaction information with the state information in the ”transaction state” queue, and perform a single write operation to the database root file. Once the I/O is complete, the queue entries are marked as having been flushed to the database root.

When a sleeping process awakens, it checks its queue entry to see if it has been processed. If the queue entry is marked as completed, the queue entry is deleted and the transaction is considered committed and final. If the queue entry has not yet processed, the queue entry is checked to see if it is the first entry in the queue; if so, this process becomes the next group poster. Otherwise, the process sleeps for a small amount of time, as before.

This algorithm continues until all "transaction state" queue entries have been processed.

5.2 New Commit Sequence

5.2.1 Previous Commit Processing

In previous releases of Rdb/VMS, when an application committed a transaction, the database root and transaction data structures were locked, modified and flushed to disk by the group commit process. The following steps were executed to commit a transaction to the database root:

1. Write the before image journal information to disk.
2. Write the database pages to disk.
3. Write the after image journal information to disk.
4. Update the root and transaction data structures.

This strategy was known as commit-to-root because the moment of commit was the updating of transaction information in the database root. This algorithm was the bottleneck for high-performance, high transaction-throughput applications. This algorithm also constrained the maximum transactions-per-second throughput that could be obtained by an application; the application became single-threaded on the root and transaction data structure locks.

With the implementation of redo logging in Rdb/VMS v4.1, step 1 has been minimized and step 2 has been eliminated. The commit-to-journal feature (described below) is designed to eliminate step 4. Once this is done, all a transaction needs to do to commit a transaction is to write its after image journal information to disk (step 3).

The sequence of operations used by step four above, which updated the root and hence actually committed the transaction, is shown in Figure 6.

Hence by avoiding this sequence, four lock requests and at least one I/O are saved per group commit. In addition, the commit sequence is not forced to be single-threaded.

1. Lock the database root information. (processes on other nodes cannot commit while this lock is held).
2. Lock the transaction information.
3. Update the database transaction information.
4. Flush the transaction information to disk.
5. Unlock the transaction information.
6. Update the database root information.
7. Flush the database root information to disk.
8. Unlock the database root lock.

Figure 6: The sequence of steps involved in committing to the database root.

5.2.2 Commit-to-Journal Feature

The database root contains a list of all active transactions. Rdb/VMS writes the database root at commit time to document that its transaction is no longer active, i.e. the transaction has completed. This is used by the snapshot facility to support read-only transactions against a transaction consistent recent version of the database. Essentially, a read-only (snapshot) transaction is given a transaction consistent view of the database AS OF the time that it begins execution. Historical versions of data are retained, stamped with the transaction id of the updating transaction that produced them.

The database root indicates to the snapshot transaction which updates it should see and which are irrelevant. It does this by retaining information as to which transactions are active when the read-only transaction starts. Updates for these active transactions are not visible to the read-only transaction. The root is the cluster-wide place to find this commit information, making it possible for all processes on all nodes of the cluster to realize a consistent view.

A new option was introduced into Rdb/VMS in order to avoid the need to write the database root. The commit-to-journal feature reduces the number of I/Os required to commit a transaction to one (the after-image journal). Note that when combined with group commit, the actual number of I/Os required to commit a transaction is usually less than one.

The commit-to-journal feature permits a user to turn off the Rdb/VMS snapshot facility in such a way that all processes accessing a database know that this feature is turned off. When snapshots are turned off in this way, Rdb need not keep the active transaction list up-to-date. Normal two phase locking provides serializability, and only the current database state is of interest. Thus, there is no need to update the database root. In addition to avoiding a write to the root, the VMS lock needed to synchronize access to the root is also avoided. In addition, the commit sequence is no longer single threaded as the root lock need not be held throughout.

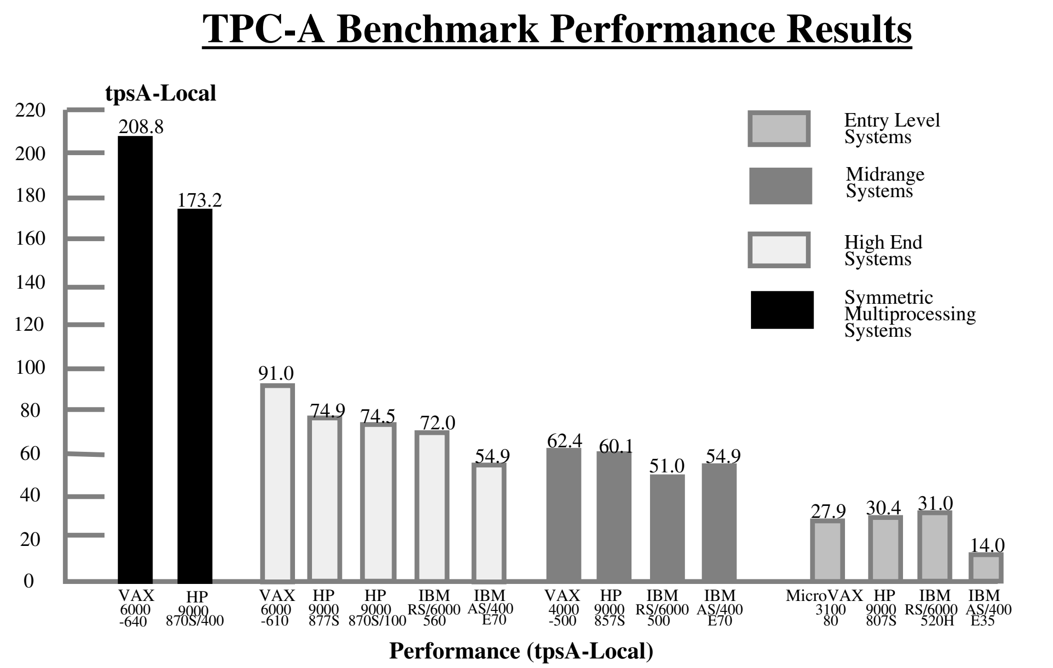
6 Discussion

The collection of features described here are the primary ones responsible for the dramatic improvement in the performance of Rdb/VMS when running the debit/credit benchmark (TPC-A [2] and TPC-B). Debit/credit performance is largely a measure of database kernel performance. Rdb/VMS performance has improved substantially when executing more complex queries, which reflect the quality of join processing and query optimization. The sources of these improvements have not been reported here, but see [1].

The systems whose performance has been reported for the debit/credit benchmark range from desktop systems to mainframes. Rdb/VMS provides its industry leadership performance and price/performance across this entire spectrum. Figure 7((a) and (b)) chart Rdb/VMS performance and price/performance on the TPC-A version of the benchmark. The TPC-A version of the benchmark has the fully loaded system including terminals and communication, etc. It is how one would normally expect to execute a debit/credit application. In each system category, Rdb/VMS performance in transactions/second or TPS, is near the top. In each category, Rdb/VMS beats its competition in $/TPS, the cost/performance metric.

The performance achieved with such a variety of system configurations clearly demonstrates Rdb/VMS's performance scalability. This scalability

[2] TPC-A and TPC-B are trademarks of the Transaction Processing Performance Council



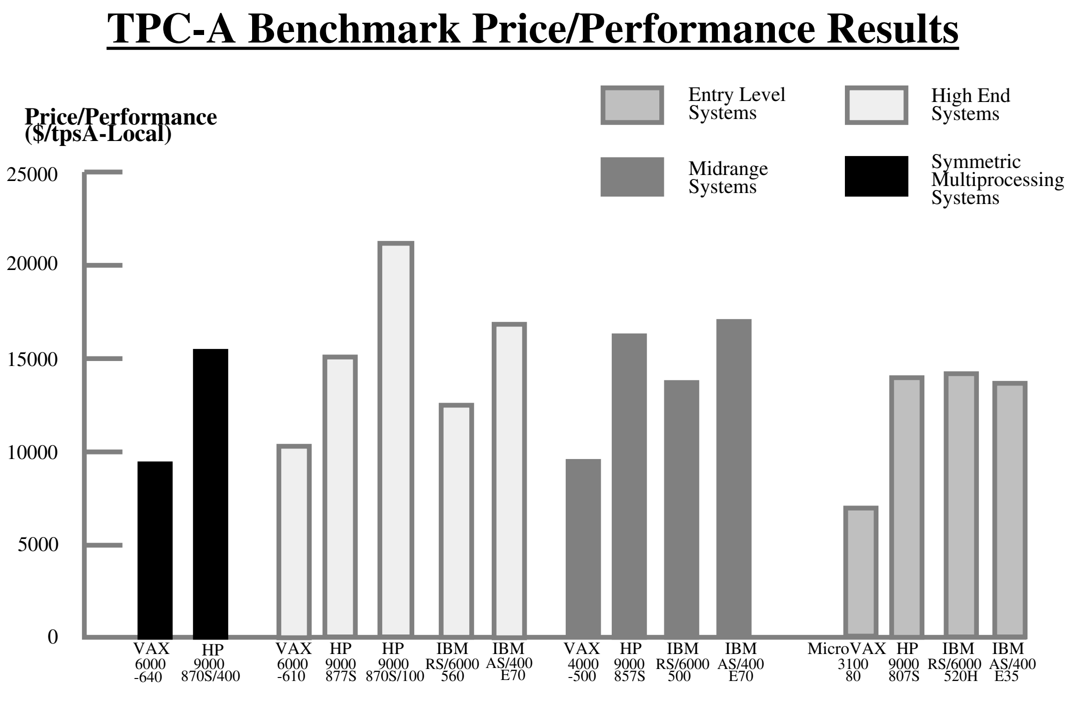


Figure 7: (a) TPC-A Benchmark Performance Results. (b) TPC-A Benchmark Price/Performance Results

encompasses processor power, I/O subsystem, disk storage, and database size. It demonstrates that Rdb/VMS will scale well in a multi-processor system and when multiple nodes of a cluster form the system configuration. The value of scalability such as this is the assurance users are given that their investment in databases are preserved as their business needs grow. That, plus its proven system robustness and the high functionality of its SQL support make Rdb/VMS the clear choice on VMS systems.

7 Acknowledgments

In addition to the authors, Annanth Raghavan, Ashok Joshi, and Jeff Arnold contributed substantially to the KODA implementation changes that produced the outstanding performance. Steve Hagan and Jay Banerjee provided management guidance and support. Improvements of the sort described in this paper are the result of a team effort, with each member making an essential contribution. We wish to acknowledge these efforts and to thank those who have helped.

References

[1] Antoshenkov, G. Dynamic Optimization of a Single Table Access. Technical Report DBS-TR-5, DEC-TR-765, DEC Data Base Systems Group (June 1991).

[2] Joshi, A. Adaptive locking strategies in a multi-node data sharing model environment. *Proc. VLDB Conf.* (Sept. 1991) Barcelona, Spain, 181-191.

[3] Joshi, A. and Rodwell, K. A relational database management system for production applications. *Digital Technical Journal* 8 (Feb. 1989) 99-109.

[4] Kronenberg, N., Levy, H., Strecker, W. and Merewood, R. The VAXcluster concept: an overview of a distributed system. *Digital Technical Journal* 5 (Sept. 1987) 7 21.

[5] Lomet, D. Recovery for shared disk systems using multiple redo logs. Digital Technical Report CRL90/4 (Oct. 1990) Cambridge Research Lab, Cambridge, MA.

[6] Rengarajan, T., Spiro, P., and Wright, W. High availability mechanisms of VAX DBMS software. *Digital Technical Journal* 8 (Feb. 1989) 88-99.

[7] Snaman, W. and Thiel, D. The VAX/VMS distributed lock manager. *Digital Technical Journal* 5 (Sept. 1987) 29-44.

[8] Spiro, P., Joshi, A., and Rengarajan, T. Designing an optimized transaction commit protocol. *Digital Technical Journal* 3,1 (Winter, 1991) 70-78.