

Building a Distributed Database with Device-Served Leases

Ohad Rodeh
orodeh@il.ibm.com

Abstract

This paper describes a method for constructing a distributed database from a set of compute-nodes, a local area network, and a set of object-disks. We assume object-disks do not fail; nodes can fail or go to sleep for long periods. In order for a node to access an object inside an object-disk a valid lease is required.

There are two issues that need resolving (1) how to build a database on top of control-units that require leases (2) handling compute-node failures.

This work extends the ARIES logging scheme to these settings.

1 Introduction

Distributed shared-disk databases [1, 2, 10] have existed for a long time. This paper describes a method for building such databases that has very good scalability and does not use clustering services [7, 18, 12]. We use device-served leases as a building block to replace group-services.

Traditional shared-disk databases are built by connecting a set of disks and a set of compute nodes together with a Storage Area Network (SAN). Group-services are run on the compute-nodes; they decide which nodes are alive and which are considered disconnected. Disconnected nodes are *fenced out* using the SAN-switch; this means that they will be inhibited from accessing the shared on-disk tables because all their requests sent through the SAN will be discarded by the switch. This is done to prevent disconnected nodes from damaging shared data. This also enforces the set of locks granted to connected nodes.

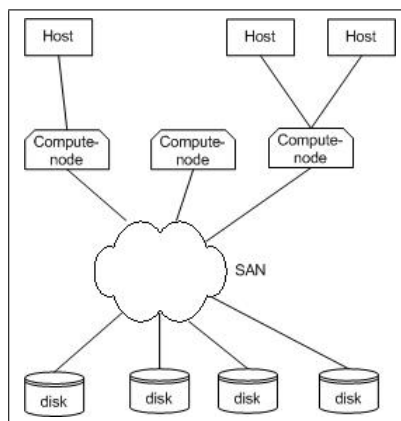


Figure 1: A traditional shared-disk database. There are compute-nodes and disks connected through a SAN. Hosts connect to the compute-nodes and receive services.

This scheme has several problems (1) it requires a switch that is able to fence out nodes (2) it relies on group-services (GCS) which are complex and have scalability limitations (3) group-services, SAN-switch, and database code need to be sown together correctly to form a complete system.

This paper suggests a novel construction which does not require group-services. The scalability limitations are overcome, and the code replacing group-services is fairly simple, connecting it to the database code is relatively straight-forward.

The scalability limitations of group-services have been dealt with in [8]. The algorithms used in GCSs are limited by the slowest node in the group; they are also sensitive to network loss rates and congestion. Furthermore, the algorithms are at least $O(n)$ where n is the number of members in the group. This means that as group size grows the GCS mechanisms work harder and harder to maintain connectivity, consistency, and reliable communication.

The system components are (1) a set of compute-nodes (2) a local area network (3) a set of object-disks [3]. An Object-Disk (OSD) is essentially a node that exports a flat file-system through a standard network protocol. A file on the OSD is called an object. We assume OSDs do not fail, handling such failures is out of the scope of this paper. We assume that the network is timed-asynchronous and compute-nodes can experience crash faults, or go to sleep for extended periods. An OSD is assumed to have some amount of non-volatile memory to speed up write commands.

Our approach should apply to most Write-Ahead-Logging (WAL) schemes. However, there is a wide variety of logging and locking schemes. We limited ourselves here to show how a specific ARIES [15] flavour can be extended to our distributed settings.

Our algorithm is aimed at low-sharing workloads; those where compute nodes access different parts of the database most of the time. It will not have good performance on high-sharing scenarios.

A database is composed of tables. We map a table to a single object on an OSD. It is possible to stripe a single table on several objects thus increasing IO throughput. We chose the simple mapping for simplicity of presentation.

We assume that we have a working single-node version of the database, DB_s . DB_s uses a specific ARIES flavour explained in section 4.1. We assume that DB_s works with object-disks and that its tables are mapped to objects. Our goal in what follows is to extend DB_s to scale to a set of compute-nodes, we shall call the multi-node database DB_m . We name our algorithm *dARIES*. A multi-user environment is assumed, each user chooses a single compute-node to work with. A user can perform the usual set of operations as with DB_s : queries and transactions. The generated workload is assumed to exhibit reasonable locality; record-sharing between nodes is relatively infrequent.

There are complexities in moving from a single-node to multiple-node database. Management and operations like bootstrap/shutdown become much more complex. This work does not address these issues; the focus is on the locking and logging schemes.

Several assumptions are made about the algorithms and behavior of DB_s . These assumptions naturally limit the generality of this work. The author believes that the variety of databases that can benefit from this work is still fairly wide. It remains for future work to lift the restrictions.

The paper is structured as follows: section 2 describes related work. Section 3 provides the reason for basing this work on object-disks rather than filers or standard disks. Section 4 describes the locking and logging algorithms used in DB_s . Section 5 describes the locking scheme for DB_m . Section 6 describes operation for DB_m during normal runs. Section 7 talks about the various failure and recovery scenarios. Section 8 discusses applications that can use *dARIES*. Section 9 summarizes.

2 Related Work

zFS [16] is a research file-system that uses OSDs. Directories and files are mapped to objects. Locking is based upon OSD-served leases. Cooperative caching is implemented. The major difference between zFS and this work is the coherency and atomicity guaranties; file-systems do not provide the ACID properties required for a database.

The Sistina file-system (GFS [4]) attempted to use disk-based locking, however, the SCSI reserve/unreserve commands that provide disk-locking were not sufficient and the attempt was abandoned.

The Palladio system [17] attempts to provide a highly scalable block-storage abstraction. The system is built out of a distributed collection of networked smart-disks and servers. As part of Palladio a comparison of different device-based locking schemes was made. For Palladio, the best scheme turned out to be a type of optimistic time-stamping. As Palladio is a block-storage system it is very different from a database and our conclusions are likely to be different.

3 Why object-disks and leases

This work could have been based on standard disks, or filers [5, 6], as long as they provide leases. The reason OSDs were chosen is that (1) the author is working on OSDs and so is aware of the possibilities of using them (2) there is ongoing work to standardize the OSD protocol, and so, there is a possibility of adding leases to it.

The reason leases are a good match for this database clustering problem is that, should a disk fail, the database cannot continue until the disk comes back up. Therefore, storing the locks on the disk separately does not improve database availability. Put another way, the disk-data and disk-locks can be thought of as a single failure domain.

4 DB_s

This section describes the locking and logging schemes for the centralized database DB_s . The descriptions are fairly brief in order to keep the presentation to reasonable bounds. For a more detailed understanding of databases the interested reader is referred to [13].

4.1 ARIES

This section describes the specific ARIES variant used for DB_s . There are many variations and optimizations to ARIES, here, a relatively simple scheme is used.

DB_s uses a log-object, referred to simply as *the log*, located on one of the object-disks, and many tables which are also OSD objects. A *page-manager* component provides locking and IO services for table pages. A typical page size, in today's databases, is 8K. It is possible to pin a page in memory; this is useful in enforcing a *write-ahead-logging* (WAL) discipline.

WAL requires writing to the log undo records for page-modifications prior to writing a page to disk, and redo entries prior to committing a transaction. This allows aborting a transaction in case of crash, and redoing it after commit. Pages need to be pinned in memory until their undo and redo logs have been written.

There is a wide variety of different logging schemes. For DB_s we assume that a log-entry refers to a single page. For example, in order to add a key-value pair to a B-tree page, assuming no splits are needed, a single log-record can be used.

In order to enforce the WAL discipline *Log Sequence Numbers* (LSN) are used. Every log entry is stamped with sequence number (LSN) provided by the log. The LSN is monotonically increasing. Each

page is stamped with the largest LSN that modified it. The journal keeps track of the largest LSN that has reached the disk, the *min-LSN*, and the page-manager needs to make sure that pages with page-LSN larger than the min-LSN are not written to disk.

ARIES is a WAL variant where the user adds redo/undo log entries for each page modification. According to the WAL discipline the page is pinned until the log-entry is written to the log. For a specific transaction log-entries are backward chained. For example, in figure 2(a) we can see a transaction with four entries: *A, B, C*, and *D*. A transaction also has *start* and *end* entries. These help the log-manager component to determine upon recovery which transaction successfully committed and which need to be aborted.

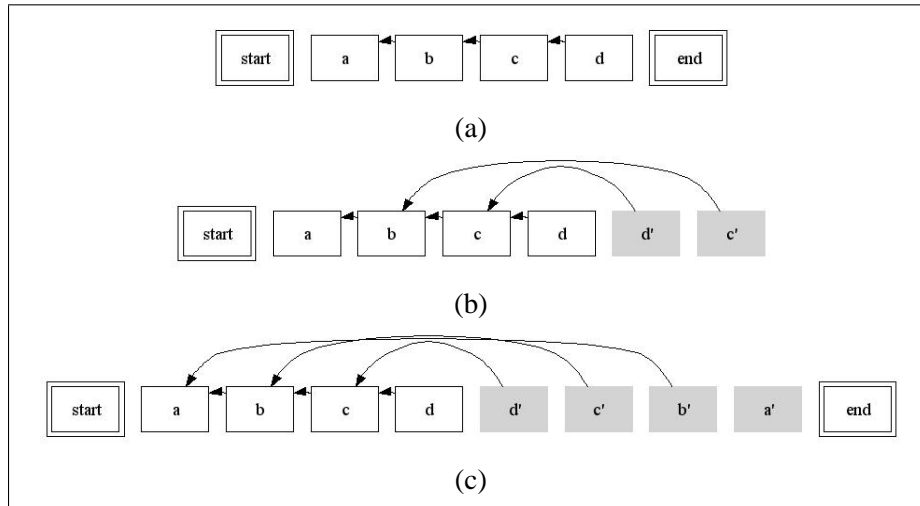


Figure 2: Log entries and CLR in a transaction. CLR are colored light-gray and marked with tags.

Transactions are aborted by user directive, or when a crash occurs. In such cases *Compensation-Log-Records* (CLR) are used. A transaction is aborted by performing its set of log-entries in undo-mode. For each log-entry that is undone a new compensation entry is added to the end of the log. The new entry has a fresh LSN and the page that is modified is marked with this new LSN; the page can be written to disk only after the CLR has reached the disk. In figure 2(b) we can see two CLR that have been added to compensate for entries *D* and *C*. The CLR points to the record previous to the one it is undoing. This helps in cases where recovery was interrupted by a crash; recovery will need to be performed again. Assuming that recovery has stopped after adding CLR for *D* and *C* then the second iteration will be able to see that *C'* points to record *B* and continue undoing *B* and *A*. This is shown in figure 2(c).

Crash-recovery is performed by a redo-pass followed by an undo-pass. During the redo-pass the log is read from beginning to end and all entries are re-done. This brings the database back to the exact point when it crashed. The undo-pass then works back from the end of the log and undoes all entries for transactions that have not committed. At the end of it the database is functional again, all effects of uncommitted transactions and partially performed transactions have been completely removed.

4.2 Locking

In order to isolate transactions from each other all transactions lock read/modified records and release them after commit. This is also called a *transaction-duration* locking discipline. A deadlock detection service identifies cycles in the lock graph and breaks cycles by choosing victim transactions and rolling them back. It is up to the application to restart aborted transactions. This may seem odd to the reader who is unfamiliar

with databases; it is legal for a user to start transactions that will deadlock each other. The database is then expected to abort one/some of the transactions and the user is expected to work things out perhaps by restarting aborted transactions.

The sections ahead discuss how to extend the algorithms from DB_s to a distributed setting.

4.3 Partial page writes

The ARIES technique we had described can get into difficulties in the presence of *partial page writes*. This is a situation where the database writes an 8K page to disk but due to a power failure only part of the page actually reaches the disk. Since disks provide atomicity on the basis of sectors (512bytes) some of the sectors may have been written while others have not. The issue is that the journal may not be able to recover from such an error.

For example, consider an 8K page P that includes 16 sectors numbered 0 through 15. There is an integer counter situated in the second sector of the page and there are log entries for incrementing and decrementing the counter. The following scenario occurs:

- A log entry E , $E_{LSN} = 4$, for incrementing the counter is written to the log.
- Page P is modified in memory
- Page P is written to disk but due to a power failure only the first sector successfully reaches the disk.

Since the LSN is in the page header it is successfully modified on disk and $P_{LSN} = 4$. However, the counter is not incremented. The ARIES recovery algorithm will not detect that log-entry E need not be executed on P and we get an inconsistency.

In order to detect partial page writes databases use checksums or some approximation of a checksum on a per-page basis. Once a partial page write is detected a rather radical action is taken: revert to an old consistent checkpoint of the database and apply all logs since.

With OSDs it might be that the object-disk will support atomic page writes. Another possibility is that OSDs will support snapshots which will allow the use of consistent checkpoints.

5 Distributed locking in DB_m

5.1 Lease support on the OSD

As a basis for locking each OSD supports a single exclusive lease that can be delegated, a *major-lease*. For OSD D , only the holder of a valid major-lease can access D . The lease is time-limited, a duration of 30 seconds is reasonable for our settings. D also records who is the lease owner. Operations allowed on the major-lease are: (1) take (2) release (3) renew. If node N_1 takes the major-lease; the OSD will not give it to other nodes. If node N_2 asks for the major-lease D will inform it that N_1 is the current owner and provide it with N_1 's network address. This provides a simple lookup mechanism for lease-owners.

Leases can be delegated. If N_1 wishes to allow node N_2 access to an object in D then it can hand N_2 a lease for D . The protocol allows N_2 to perform an operation as long as a timely lease is attached to the request. For example if $lease_D$ is bounded between t and $t + 30$ then N_2 will be able to perform operations on D as long as the local time at the target when the request is received is in the range $\{t...t + 30\}$.

The major-lease is implemented as a 64bit number; it is the local time on the OSD at the time it is given. For example, if a major-lease is requested at time 3003 on the OSD then the major-lease is going to be 3003. Each request is verified for timeliness of the major-lease (m_j) attached to it by checking the

difference between m_j and the current-time. For example, if the current time is 3003 then a request with lease 2990 will be accepted; a request with lease 2950 will be rejected.

It is assumed that compute-nodes are non-malicious; a malicious node can invent lease times and circumvent the major-lease protection mechanism. Major-leases replace the fencing mechanisms used by traditional databases.

5.2 Locking for records, pages, and tables

A table is composed of records, and a database needs to provide record-level read/write locking. Several records can fit into a single page, which is the smallest IO unit. If two nodes lock and modify two records on the same page, conflicts will arise. Therefore, in this work we provide distributed locking on the granularity of pages and tables, not records. Note, however, that once a node locks a page it can internally provide record-locks to its transactions. To avoid confusing the reader we shall ignore the distinction between records-locks and page-locks. We shall say that in order for a transaction T on node A to modify record R which is located on page P it needs to lock page P . In reality this will translate into node A taking a page lock on P and providing a finer-grained record-lock for R to transaction T .

In order to perform locking we take an approach similar to zFS [16]. For each OSD we run a lock-manager (LKM). The manager provides a page-level and object-level lock-service to all database components. The lock-manager for OSD D , denoted by D_{LKM} , can run on any compute-node. D_{LKM} operates by taking the major-lease for D and continuously renewing it.

Compute-node N_1 that wishes to take a lock on a page/object on D locates D_{LKM} by querying D for its lock-manager. N_1 then creates a connection to D_{LKM} . The connection can be implemented by a TCP network connection over which protocol messages can be reliably passed. As long as the connection is alive there is no need for further lookup requests from N_1 to D . Through the connection N_1 can take and release read/write locks on objects and pages. The protocol includes messages for the client to (1) take a lease on the server (2) renew it (3) release it. This lease is different than the OSD major-lease; it protects the client-server protocol between lock-taker and lock-manager. Upon connecting to D_{LKM} node N_1 takes a lease and henceforth renews it periodically. As long as N_1 's lease is valid all of N_1 's locks will be respected. When the lease is broken D_{LKM} assumes that N_1 has failed. It will wait until the lease expires and then notify nodes that wish to take locks on areas previously locked by N_1 that recovery needs to be performed.

D_{LKM} provides a valid major-lease for OSD D to all of its clients as part of the connection protocol. This allows connected clients to access D directly. Clients that get disconnected, due to a network problem, from the lock-manager will be able to access the OSD until their major-lease expires. We assume clients are not malicious and that they access only pages they had previously locked; the major-lease mechanism will not protect the OSD from malicious accesses.

A lock server needs to guarantee that even if it fails, locks it has granted will be respected. This can be done either by making the lock-manager highly available or by recording on disk all granted locks (*hardening*). For this exposition we chose the second method. Server D_{LKM} creates an object D_{locks} on D . This object contains the list of all currently given locks. D_{locks} is updated whenever locks are granted or released. Access to D_{locks} is possible only to the current D_{LKM} because only it holds a valid lease to D . Should D_{LKM} fail another compute-node will take its place after the OSD lease expires; the new server will recover the locks from D_{locks} . Hardening the locks is not very expensive because the OSD has a battery backed write-cache which makes writes fairly quick.

5.3 Deadlocks

Providing page, and table locking is good but not sufficient. Deadlocks can happen because two nodes N_1 , and N_2 request write-locks on pages P_1 and P_2 in reverse order; N_1 ends up with P_1 requesting P_2 and N_2

holds on to P_2 while requesting P_1 . Longer cycles can occur as well. The problem is that there is no global lock manager with knowledge of the set of taken and requested locks.

If DB_s follows strict two-phase locking, or takes locks in lexicographic order then deadlock avoidance can be implemented locally. However, this is not a valid assumption as most databases use much more relaxed locking schemes.

The problem, in essence, is that the deadlock handling algorithm from DB_s needs to be ported to DB_m . On each compute node we run the standard DB_s deadlock-handling module to resolve local problems. A simplistic solution to global deadlock detection can be:

1. Each compute-node N , once in a while, queries the set of compute-nodes for their set of on-going transactions. The set of compute-nodes known to N can be approximate; as long as all live nodes will eventually get into this set.
2. Node N constructs a graph and checks for cycles
3. In case a cycle is detected N chooses a victim transaction T and sends a message to the node running it abort T

This solution can sometimes make mistakes and abort transactions unnecessarily; this is considered tolerable because the user needs to handle aborted transactions anyway. There are much better algorithms than this one, however, this topic is out of the scope of this paper.

5.4 Examples

Before going into the next sections a few examples can help clarify the use of leases. Assume that there is one OSD: D and two compute-nodes: $\{N_1, N_2\}$. Initially, the major-lease for the disk is not taken.

Scenario where node N_1 reads page P_1 from disk D :

1. N_1 needs to lock page P_1 on disk D .
 - (a) N_1 takes the major-lease for D and creates a local lock-manager for D .
 - (b) N_1 gets a read-lock for P_1 from D_{LKM} .
 - (c) N_1 sends a read request for page P_1 to D ; the major-lease is attached.
 - (d) D sends the page to N_1 .
2. Every 20 seconds D_{LKM} renews its major-lease from D .

Continued scenario where node N_2 reads page P_1 from disk D :

1. N_2 contacts D and requests the major-lease, D informs N_2 that node N_1 holds the major-lease.
2. N_2 creates a connection to the lock-manager on N_1 , N_1 sends back a timely major-lease for D . From now on, every time D_{LKM} renews the major-lease it sends it to both N_1 and N_2 .
3. N_2 gets a read-lock on P_1 from D_{LKM} .
4. N_2 sends a read request for page P_1 to D ; the current major-lease is attached.
5. D sends the page to N_2 .

Node N_1 disconnects for more than 30 seconds:

1. N_2 cannot renew its lease on its connection to D_{LKM}
2. After 30 seconds N_2 loses the ability to read/write from D because it no longer has a valid lease.
3. N_2 takes the major-lease on disk D and creates a local lock-manager for it.

The time-asynchronous model assumes that clock drifts between nodes in the cluster are bounded. So, while nodes may not agree on the current time, they drift apart at a bounded rate.

The timed-asynchronous assumption is used for coordinating the leases. A lock-manager for disk D assumes that a compute-node N_1 that has been disconnected for more than 30 seconds will not be able to access D . Disk D will presumably reject requests from N_1 because they come with an old lease. However, if the clock on the lock-manager moves much quicker than the clock on the disk then N_1 will still be able to read/write from D while its locks are revoked and handed to other nodes.

6 Normal runs for DB_m

In DB_s the compute-node has a log. Similarly, for DB_m each node creates a log-object for itself on one of the OSDs and takes an exclusive lock on it. The log-object is used to store the write-ahead-log for the compute node. Each node accesses only its own log-object, for node N_1 the log object is denoted by log_{N_1} . In case N_1 dies another compute node will be called to recover log_{N_1} . Access to the log is protected with an exclusive lock.

Each compute node runs a set of transactions requested by hosts connected to it. Each transaction is executed by exactly one node; transactions are never sliced into pieces where different nodes perform different parts. This also means that for each transaction there is exactly one log object where all of its entries are located.

As part of the lock-manager connection protocol the client node declares where its log is located. When the manager notices that a client lease has expired without being properly released it assumes the client has crashed. This means that all client locks cannot be granted to another compute-node until recovery is performed on locked-areas.

To perform a transaction in DB_s the (only) compute-node needs to:

1. Take locks on the pages.
2. Write an open-transaction entry to the log.
3. Add log-entries to the log and modify the records in-memory.
4. Write a close-transaction entry to the log.
5. Release the page locks.

We use, in essence, the same algorithm in DB_m . Assume compute-node N_1 is performing a transaction T , the pages in the transaction are on OSDs D_1 and D_2 .

1. Connect to the lock-managers for D_1 and D_2 , request locks on the pages.
2. Write an open-transaction entry into log_{N_1} .

3. Add log-entries to log_{N_1} for each record modification.
4. Write a close-transaction entry into log_{N_1} .
5. Release the page locks. N_1 may choose to hang on to the page locks as long as other nodes do not request them. This allows write-back caching. When N_1 does decide to release a page lock it first needs to write the page to disk.

There are techniques that allow nodes to pass modified pages and records between them. We chose not to attempt that here. In our settings the lock-manager maintains one log-pointer per page. A compute-node that will attempt to take a page-lock, modify a page, and release the lock without writing it to disk first can cause a consistency problem. For example, take the following scenario:

- (a) Node N_1 takes locks for pages P_1 and P_2 .
- (b) N_1 modifies pages P_1 and P_2 .
- (c) N_1 writes P_1 (but not P_2) and releases the locks.
- (d) Node B takes the lock for P_2 .
- (e) Node N_1 fails.

No one will know that there is a log-entry to replay for P_2 and N_1 's updates will have been lost. As P_2 was part of a transaction where some of the effected pages were written to disk and some were not, the database is now in an inconsistent state.

6.1 A single-LSN per-page

In DB_s each page is stamped with an LSN where LSNs are chosen by the log component. The LSN synchronizes between the database cache and the log. In DB_m each compute-node has its own log-component that issues independent LSNs. There needs to be some way to synchronize between these logs so that a single coherent LSN will be attached to each page. For example, there is a requirement that the LSN on a page will grow monotonically.

Lets examine what can happen in the absence of LSN synchronization.

1. Node N_1 takes a write-lock for page P .
2. Node N_1 modifies P and marks it with LSN 10.
3. Node N_1 writes P to disk and then releases the lock.
4. Node N_2 takes the write-lock for P .
5. Node N_2 modifies P and marks it with LSN 6.
6. Node N_2 writes P to disk and then releases the lock.

Clearly, if N_1 ever takes the lock for P again, and then fails and recovers it will redo the modification for LSN 10. This is because the LSN is now 6. Redoing the log is an error because ARIES log-entries are not idempotent, replaying an entry twice is erroneous.

Our solution is similar to the one presented in [11]; it is also reminiscent of Lamport time-stamps [14]. Node N_1 , when it reads a page P , sets its local LSN to the maximum between the current LSN and P 's LSN. This way, when P is modified it will contain an LSN that is strictly larger than before. This is legal

because for ARIES to work a compute-node's LSN needs to grow monotonically, gaps in the LSN sequence can be tolerated.

Caution is needed here because some databases use the LSN as more than a simple counter. The LSN value is used to mark where the log-entry is located in the log-file. A possible solution is to enlarge the LSN for DB_m and include more information. This has the disadvantage of modifying the page-format.

6.2 Rollback

DB_s supports rollback. There are at least two cases in which rollback is needed (1) the user wants to abort a transaction (2) a deadlock was detected and one of the transactions needs to be rolled back.

To support rollback in DB_m we emulate the DB_s solution. Assume node N_1 is performing a transaction T . Initially N_1 holds (1) a set of locks protecting the set of modified pages (2) a lock on log_{N_1} . To rollback N_1 will:

1. Perform the set of log-entries for T in undo mode and add a CLR to log_{N_1} for each modification.
2. For each modified page N_1 will:
 - If the page isn't in cache then read the page from disk.
 - Modify the page.
 - Write the page to disk.
3. Releases all page locks.

Since N_1 holds all page-locks initially then no deadlocks can happen during rollback.

7 Recovery

There are several failure scenarios to consider. We first discuss how simple lease expires are handled. The simplest and most common problem is when a client loses its lease on a lock-manager. This might mean that pages it has locked on a disk will be given over to another node. Lets assume that the lease-manager is for OSD D . The protocol to handle such cases has two types of players (1) the original owner node N_1 (2) Another node N_2 that takes a page-lock on a page P previously owned by N_1 .

Once N_1 loses its lease it checks whether it still has the lease on Log_A . If so, it will re-acquire a lease on D from D_{LKM} , and continue as usual. If N_1 has lost its lease on Log_A then full recovery is needed; N_1 breaks all its lock-manager leases and leaves all of its transactions in mid-flight. Full recovery is needed.

When N_2 takes a page-lock on P it gets a notification that recovery is needed for P and the log is located at Log_A . N_2 attempts to take the lease on Log_A . If successful, it needs to recover N_1 's log, see the next subsection. Otherwise, N_1 is still alive and holding the lock for its log; N_2 releases the page-lock on P because N_1 will shortly recover, renew its lease on D , and continue processing. N_2 needs to wait until N_1 correctly releases its page-lock on P .

7.1 Recovery from a compute-node failure

If node N_1 fails and recovers it needs to replay log_{N_1} . Log recovery is performed by taking the exclusive lock on log_{N_1} and performing a redo and then an undo pass. A log-entry E that applies to record R in page P is replayed by the following sequence: (1) take the lock P (2) check if P_{LSN} is lower than E_{LSN} (3) If so then apply the update to P and update P_{LSN} . CLRs are added to log_{N_1} according to ARIES.

After N_1 completely recovers it erases the *need recovery* mark from its pages. In the event of partial recovery, the marks will not be removed. This sounds counter-intuitive because it locks pages for longer than needed in the need-recovery state. However, it ensures that all dirty-pages belonging to a node that has crashed will remain marked on disk until full recovery is achieved. This also ensures that there will be no deadlocks during recovery.

The lock-manager will grant locks to nodes for pages that have previously been locked by a failed node only after the failed-node's lease expires. It will notify the requester that it needs to recover the page and provide the failed-node's log object name.

If a node fails and does not recover then other nodes will be stuck waiting for its transactions to complete. This is a serious problem in a distributed environment because nodes can become disconnected, slow, or suffer from slow network connections. To solve this, node N_1 can replay the log for node N_2 if node N_2 loses the lock for log_{N_2} . Recovery by node N_1 of log_{N_2} is similar to recovery by the owner node; N_1 performs recovery while holding log_{N_2} 's exclusive lock so it cannot be interrupted.

Once a transaction's commit record is written to disk a transaction is ensured of success. Even if the initiating node fails all modified records are still locked. Any node that stumbles upon any of these records will be told to perform recovery on behalf of the failed node. It will then replay the transaction from the initiator's log.

7.2 Recovery from a lock-manager failure

Compute nodes can fail taking down with them all lock-managers located on them. If the lock-manager for OSD D fails it cannot be replaced until the OSD lease it took expires. Connections to failed lock-managers are torn down and lock holders know that lock-manager recovery will take place.

After the major-lease for D expires another compute-node will take the major-lease for D and create a local lock-manager (D_{LKM}). D_{LKM} recovers the set of granted locks from object D_{locks} . It pessimistically assumes that all lock-holders have also crashed and notifies all lock-requesters for previously locked areas that recovery may be required.

7.3 Recovery from multiple failures

There are several cases here:

Several compute-nodes fail: Since dependent transactions are disallowed failure of several compute-nodes simply requires recovering their logs separately. Our scheme requires a page to be written to disk before changing compute nodes. Therefore, there can be at most one log object with unapplied entries for a page.

Several lock-managers fail: There are no inter-dependencies between lock-managers, so recovery is just to recover each lock-manager separately.

Several compute-nodes and several lock-managers fail: As compute-nodes depend on the services of the lock-managers then lock-managers need to be recovered first.

7.4 High availability

DB_m can be categorized as a system that supports lazy recovery. Any compute-node can fail taking down with it not only the set of transactions it is currently executing but also the lock servers running on it. Such a failure can go on undetected for a long time; the next node to access data last touched by the failed node will

bear the burden of recovery. Recovery will include replaying the log as well as reviving any lock-servers required.

It is possible to convert this lazy approach to recovery into a more active one. This can be done by actively replicating the lock-servers and by having each node search for failed nodes once in a while. When failed nodes are found their logs will be replayed on their behalf.

8 Applications

dARIES will probably not be competitive with a large SMP machine that runs a database because high-sharing workloads will simply run a lot slower. However, there are important cases of low-sharing. Databases that are used mostly for searching are good candidates.

Modern file-systems use a database structure to store their meta-data: file-metadata and directory structure. For example, the StorageTank [9] file system splits the file-system hierarchy into a set of sub-trees. A server-node is in charge of all modifications/queries to this sub-tree.

There are three types of nodes in the systems: hosts, servers, and network disks; see figure 3. All are connected to a fast network. The servers are also known as *Meta Data Servers* or MDSs.

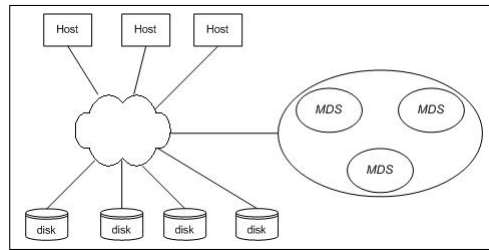


Figure 3: A simplified picture of storage tank.

Hosts get meta-data from servers and perform IO to disks. To read a file a host get an extent list from the MDS responsible for the file and then reads the file directly from disk. To modify a directory the host requests the MDS to perform the directory operation on its behalf. Group services are used to keep track of the set of live servers. If a server fails another server takes its place by taking over the database used for the directory sub-tree. Hosts need to keep in touch with their MDS cluster, otherwise they are assumed to be dead and their locks are revoked. To prevent such hosts from accessing the disks and creating havoc the MDSs fence them out by telling the network fabric to ignore all disk-IO requests from “dead” hosts.

Storage-Tank is a SAN file-system and the nice property it has is allowing hosts to perform IO directly to disk while keeping the meta-data servers out of the data path. The interesting point here is that the MDS database system can be made to use dARIES. One can locate the sub-trees on separate OSDs, and use a compute-node for each MDS. The workload is almost always local because most transactions occur within a sub-tree.

9 Summary

dARIES is expected to work well if there is limited sharing between nodes. If there is a lot of locality between nodes and OSDs then pages and locks can be cached for extended periods by compute-nodes. Behavior should approximate ARIES on a single-system with a local disk.

The major assumption made here is the use of novel storage-devices, OSDs, that support a particular form of locking. We expect object-disks to become prevalent in the future and therefore trying to utilize them is important.

The upshot of using OSD-served leases is that group-services are no longer required. This leads to a system design that has horizontal scalability. As more OSDs are added, more tables can be added leading to a distribution of IO across more backend disks. As more compute-nodes are added users can be more widely spread around to reduce CPU load on compute-nodes. Assuming good locality, lock-managers will serve the subset of the users that access a specific OSD so their load will not be high.

The constraints assumed about DB_s are that (1) it uses page-based ARIES and log-sequence numbers (2) deadlock detection instead of deadlock avoidance. It remains for future work to attempt lifting these limitations.

It is the author's opinion that this technique can lead to interesting new distributed database designs.

10 Acknowledgments

The author would like to thank: Gary Valentin who helped a lot with ARIES and databases in general, and Avi Teperman who did related file-system research on zFS. Thanks is also due to Itai Segall and Effi Ofer who helped review the initial drafts.

I would also like to thank other people with which I've had discussions about the above ideas: Mark Hayden, Roy Friedman, C. Mohan, Dalia Malkhi, Inderpal Narang, Robbert Van Renesse, and Paula Ta-Shma.

References

- [1] www-306.ibm.com/software/data/db2.
- [2] www.oracle.com.
- [3] www.snia.org/tech_activities/workgroups/osd.
- [4] www.redhat.com/software/rha/gfs.
- [5] www.netapp.com.
- [6] www.emc.com.
- [7] *Group Services Programming Guide and Reference, RS/6000 Cluster Technology*. IBM, International Technical Support Organization, 2000.
- [8] Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.
- [9] R. C. Burns. *Data Management in a Distributed File System for Storage Area Networks*. PhD thesis, University of California, Santa Cruz, March 2000. <http://www.almaden.ibm.com/cs/storagesystems-/stortank/rbdissert.pdf>.
- [10] C., Mohan and Inderpal, Narang. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Very Large Databases (VLDB)*, 1991.

- [11] C. Mohan and Inderpal, Narang. Data base recovery in shared disks and client-server architectures. In *ICDCS*, pages 310–317, 1992.
- [12] Hayden, M. The Ensemble system. Phd Thesis TR98-1662, Cornell University, Computer Science, 1998.
- [13] Jim Gray and Andreas Reuter. *Transaction Processing: concepts and techniques*. Morgan Kaufmann Publishers, Inc., 1993.
- [14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. 21:558–565, July 1978.
- [15] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. 17(1), March 1992.
- [16] O. Rodeh and A. Teperman. zFS – a scalable distributed file-system using object-disks. In *Goddard Conference on Mass Storage Systems and Technologies*, April 2003.
- [17] Richard Golding and Garth Gibson. Highly concurrent shared storage.
- [18] Stanton, J. and Amir, Y. The Spread wide area group communication system. TR CNDS-98-4, Center for Networking and Distributed Systems, Computer Science Department, Johns Hopkins University, 1998.