

M. Tech. Dissertation

Transaction Management in Parallel Databases

submitted in partial fulfillment of the requirements for the degree of
Master of Technology

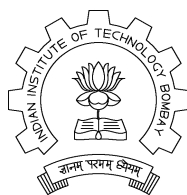
by

P.P.S. Narayan

95305401

under the guidance of

Dr. S. Seshadri



Department of Computer Science and Engineering

Indian Institute of Technology

Mumbai

January 27, 1997

Dissertation Approval Sheet

This is to certify that the dissertation titled **Transaction Management in Parallel Databases** by **P.P.S. Narayan** is approved for the degree of **Master of Technology**.

Dr. S. Seshadri
(Guide)

Internal Examiner

External Examiner

Chairman

Date : _____

Abstract

As applications require higher storage and easier availability of data, the demands are satisfied by better and faster hardware. Parallel architectures are exploited by introducing *data* and *program* parallelism in database systems. But, with multiprogramming environments that use CPU and I/O parallelism, maintaining the correctness of the database state is important.

Transaction management ensures the *atomicity*, *consistency*, *isolation* and *durability* properties of applications running concurrently on the database system. The main components of transaction management are concurrency control, that ensures the isolation, and recovery, which ensures that changes to database state are consistent, atomic and durable.

In parallel architectures, the existence of multiple nodes performing operations on the disks of the database leads to new issues in transaction management. Concurrency control and recovery needs to be performed across nodes in the system.

In this dissertation, we propose restart recovery algorithms for system-wide failure and single-node failure, along with the fuzzy checkpointing protocol. The methods work with the *steal* and *no-force* cache management policies. In fact, the pages can carry dirty updates between nodes without being flushed to disk. We have an implementation of the recovery protocols and have tested the implementation as ARIES on a single-site, as well as for multi-node restart recovery.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Introduction to Parallel Databases | 1 |
| 1.2 | Parallel Architectures for Database Systems | 2 |
| 1.3 | Parallelism in Database Systems | 2 |
| 1.4 | Introduction to Transaction Management | 3 |
| 1.5 | Scope of Work | 3 |
| 1.6 | Overview | 3 |
| 2 | Review of Literature | 5 |
| 2.1 | Logging, Savepoints and Checkpointing | 5 |
| 2.2 | ARIES | 7 |
| 2.2.1 | Updates | 7 |
| 2.2.2 | Total or Partial Rollbacks | 8 |
| 2.2.3 | Restart Recovery | 8 |
| 2.2.4 | Parallelism during Restart Recovery | 9 |
| 2.3 | Multi-Level Transactions and Recovery | 9 |
| 2.4 | Recovery in a Shared-Disk Environment | 10 |
| 2.4.1 | Distributed Logging | 10 |
| 2.4.2 | LSN with Multiple Logs | 11 |
| 2.4.3 | WAL Protocol in a SD Environment | 11 |
| 2.4.4 | Checkpointing and Restart Recovery | 12 |
| 2.5 | Reduced Locking and Latching Using LSNS | 12 |
| 2.6 | Summary | 13 |
| 3 | Software Architecture of <i>Brahmā</i> | 14 |
| 3.1 | Modules of <i>Brahmā</i> | 14 |
| 3.1.1 | Operating System Layer (OS Layer) | 14 |
| 3.1.2 | Cache Manager (CM) | 15 |
| 3.1.3 | Information Processor (IP) | 15 |

| | | |
|----------|--|-----------|
| 3.1.4 | Storage Manager (SM) | 15 |
| 3.1.5 | Object Manager (OM) | 16 |
| 3.2 | New Modules of <i>Brahmā</i> | 16 |
| 3.2.1 | Transaction Manager (TM) | 16 |
| 3.2.2 | Log Manager (LGM) | 16 |
| 3.2.3 | Recovery Manager (RM) | 17 |
| 3.2.4 | Checkpoint Manager (CKM) | 17 |
| 3.3 | Interaction between the Modules | 18 |
| 3.4 | Summary | 18 |
| 4 | Overview of Recovery in <i>Brahmā</i> | 19 |
| 4.1 | Normal Processing | 19 |
| 4.1.1 | Changes to DPT Entries | 19 |
| 4.1.2 | Cache Coherency | 20 |
| 4.2 | Checkpointing | 21 |
| 4.3 | Restart Recovery from System-Failure | 21 |
| 4.4 | Restart Recovery from Single-Node Failure | 22 |
| 4.5 | Summary | 22 |
| 5 | Transaction Manager | 23 |
| 5.1 | Components of the Transaction Manager | 23 |
| 5.1.1 | Transaction Information | 23 |
| 5.2 | API of the Transaction Manager | 24 |
| 5.2.1 | For Recovery | 24 |
| 5.2.2 | Computing CommitUSN | 25 |
| 5.3 | Summary | 25 |
| 6 | Log Manager | 26 |
| 6.1 | Logging | 26 |
| 6.1.1 | Log Address and Extents | 26 |
| 6.1.2 | Disk Log Header | 27 |
| 6.1.3 | Initialization of Log Manager | 28 |
| 6.2 | Types of Logs | 29 |
| 6.3 | Log Cache | 31 |
| 6.4 | Writing Logs | 33 |
| 6.4.1 | Splitting Big Logs | 35 |
| 6.5 | Log Flush | 36 |
| 6.6 | Log Iterator | 37 |

| | | |
|-----------|--|-----------|
| 6.7 | Special Calls | 38 |
| 6.8 | OverWriteUSN | 39 |
| 6.9 | Summary | 40 |
| 7 | Examples of Logging | 41 |
| 7.1 | In the Object Manager | 41 |
| 7.1.1 | Object Creation | 41 |
| 7.1.2 | Object Deletion | 43 |
| 7.1.3 | Object Updation | 44 |
| 7.2 | In the Object Storage Manager | 45 |
| 7.3 | Summary | 47 |
| 8 | Checkpointing Protocol | 48 |
| 8.1 | Checkpoint Coordinator | 48 |
| 8.2 | Checkpoint Manager (CKM) | 49 |
| 8.2.1 | Issues in Checkpointing | 50 |
| 8.2.2 | Improving the Algorithm | 54 |
| 8.3 | Summary | 54 |
| 9 | Recovery from System-Failure | 57 |
| 9.1 | Recovery Coordinator | 57 |
| 9.1.1 | Distributing Redo Work | 59 |
| 9.2 | Summary | 60 |
| 10 | Recovery Manager | 61 |
| 10.1 | Initialization | 61 |
| 10.2 | Transaction Rollback | 61 |
| 10.3 | Restart Recovery | 62 |
| 10.3.1 | Analysis | 62 |
| 10.3.2 | Redo | 62 |
| 10.3.3 | Undo | 64 |
| 10.4 | Summary | 64 |
| 11 | Recovery from Single-Node Failure | 65 |
| 11.1 | Assumptions | 65 |
| 11.2 | Detection of Failure | 65 |
| 11.3 | Information Processor | 66 |
| 11.4 | Algorithm | 67 |
| 11.4.1 | Discussion | 68 |

| | |
|---|-----------|
| 11.4.2 LOGTYPE_PAGEFETCH Log | 70 |
| 11.5 Summary | 71 |
| 12 Implementation and Tests | 72 |
| 13 Conclusion and Future Work | 74 |
| 13.1 Comparisons with Related Work | 74 |
| 13.2 Overview | 75 |
| 13.3 Future Work | 76 |
| 13.3.1 Transaction Manager | 76 |
| 13.3.2 Single-Node Failure Recovery | 77 |
| 13.3.3 Lock Manager | 77 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Savepoint and CLRs | 6 |
| 2.2 | ARIES Model and Data Structures | 7 |
| 3.1 | Modules of <i>Brahmā</i> | 17 |
| 6.1 | Sequential Log Extent | 27 |
| 6.2 | Disk Log Header and Preallocated Log Extents | 28 |
| 6.3 | Log Types used in <i>Brahmā</i> | 30 |
| 6.4 | Split Logs put in Log Extents | 36 |
| 8.1 | Page reaching a node after checkpoint at both nodes | 51 |
| 8.2 | Page is shipped and disposed before the sender's checkpoint but reaches the requester after its checkpoint | 52 |
| 8.3 | Checkpointing Protocol | 55 |
| 9.1 | Restart Recovery Protocol | 58 |
| 11.1 | Lost Information of the IP on crash | 66 |

Chapter 1

Introduction

1.1 Introduction to Parallel Databases

In recent years, the amount of data being stored in database systems (DBS) has increased drastically. At the same time, customers using large databases want data to be readily available.

Furthermore, there are new and complex applications in CAD, GIS and Multimedia where data volume is large as compared to conventional applications. Under such circumstances, enhancements to query languages like SQL and 4GL for more interactive and complex queries is expected. Therefore, more logic will be pushed into the DBS for better data sharing and performance.

One way to improve the performance of DBS is to provide faster processors and larger memory at the hardware level, keeping the same software environment. Some limitations of this would be (a) the higher cost per MIPS of faster machines as compared to parallel machines built from smaller processors, (b) the increasing memory requirement of new applications, and (c) increasing gap between the CPU speed and I/O bandwidth.

In multi-processor architectures CPU parallelism is used to increase the computing power and I/O parallelism is employed to reduce the gap between the processor speeds and I/O bandwidth. Aggregate memory attached to multiple processors will be able to satisfy the memory requirements of new applications.

Conventional DBS can provide the storage capability for any kind of data. But, with developments in parallel architectures, the DBS could be adapted to improve the availability and storage of data. The commercial success of various parallel databases vendors like Teradata, Tandem, Oracle and NCR are testimony of the fact that it is more than just a research curiosity [DG92].

1.2 Parallel Architectures for Database Systems

Capacity and availability of a single-processor DBS can be improved by using multiple processors. The various architectures for this are [DG92][MPTW94]:

- **Shared Nothing (SN) or partitioned data:** With SN, each processor owns a portion of the database and only that portion can be directly read or modified by that processor. The transactions accessing data in multiple processors will require a two-phase commit protocol to coordinate its activities. This architecture reduces interference by minimizing resource sharing, thus leading to high scalability.
- **Shared Disk (SD) or data sharing:** With SD, all disks containing the databases are shared among the different processors and each processor has its own main memory. Every processor performs actions that may access and modify the databases on the shared disks. Since the same data may be accessed by the different processors concurrently, the interaction is controlled by synchronization protocols. Cache coherency protocols degrade performance considerably and the number of shared resources limit its performance.
- **Shared Everything (SE):** In SE, memory in addition to the disks, is shared across the processors. Contention for accessing shared resources, like disks and memory, is a major problem in this architecture and it may impede high scalability.

1.3 Parallelism in Database Systems

Parallelism within the DBS can be achieved in two ways:

- **Program Parallelism**, where the execution of multiple operations of a given program is done in parallel. A pipeline between tasks is one mechanism of achieving program parallelism.
- **Data Parallelism**, where the execution of a single operation is done on different pieces of the input data in parallel. Data parallelism is achieved by partitioning the data on a number of disks.

Both program and data parallelism are possible in all the three architectures: SN, SD and SE. A mixed approach of data and program parallelism is also an alternative.

1.4 Introduction to Transaction Management

A transaction is a collection of operations on the physical and abstract database state [GR93]. Many of the concepts of transaction management have been pioneered from distributed and fault-tolerant computing. The ACID properties of transactions, which have emerged as the unifying concepts of distributed computing, are listed below:

- **Atomicity:** A transaction's changes to the database state are atomic; the all or none property.
- **Consistency:** A transaction is a correct transformation of the state. The actions taken as a group do not violate the integrity constraints associated with the state.
- **Isolation:** Even though transactions execute concurrently, it should appear as though no two transactions executed together. That is, for each transaction T, it appears that others executed before T or after T.
- **Durability:** Once a transaction completes successfully its changes to the state must survive failures.

Atomicity, consistency and durability are addressed by recovery, while isolation is enforced by concurrency control.

1.5 Scope of Work

The scope of this project is to introduce recovery into “*Brahmā*”, the parallel database system being developed on a *Shared-Disk* parallel machine, *Anupam*. Details of the *Anupam* system can be obtained from [MDRK94].

1.6 Overview

Chapter 2 presents some of the related work done in recovery and discusses some issues of recovery in a parallel architecture. The existing implementation of *Brahmā* is described in *Chapter 3* along with the introduction of new recovery components.

A general overview of the recovery algorithm is presented in *Chapter 4*. The new components added to *Brahmā* for recovery have been described in *Chapter 5*, *Chapter 6*, *Chapter 10* and *Chapter 8*. Examples of logging are described in *Chapter 7*.

A detailed description of the protocol for restart recovery from system-failure and the recovery coordinator are presented in *Chapter 9*. Extensions to handle single-node failures in the system are presented in *Chapter 11*. Implementation and testing details are given in *Chapter 12*. Comparisons with related work and salient features of our design are discussed in *Chapter 13*. Future extensions are also suggested in it.

Chapter 2

Review of Literature

The recovery subsystem in a database system provides the mechanism for atomicity, consistency and durability. Atomicity is provided by undoing aborted transactions, redoing committed transactions, and coordinating transactions that are distributed. Recovery provides consistency by aborting the transactions that fail integrity constraints on completion. Forcing of logs to stable storage at commit, along with redo processing at restart, ensures durability.

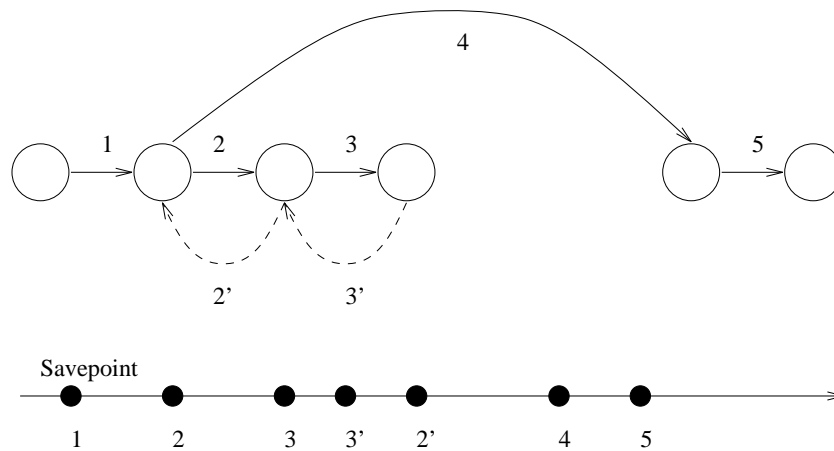
In this chapter, we will introduce some of the terms used in recovery. We assume familiarity with basic recovery terminology as described in [KS91, EN94]. Also presented in this chapter are commonly used recovery algorithms, along with a discussion of the issues concerning recovery in a shared-disk architecture.

2.1 Logging, Savepoints and Checkpointing

Log is a sequence of records which documents changes to the database state. Log records appear – (a) for each update to the database, (b) to indicate events like the beginning or termination of a transaction, and (c) as information to reduce log scans during recovery. Each log record has an unique identifier called its *log sequence number* (LSN), which can be seen as a logical address of the log on the log disk. The LSN needs to be monotonically increasing number.

A log record can be either to *redo* the update action, to *undo* the logged action or to do both. The logged information may be in terms of the *physical* before-after images of the change or in terms of the *logical* database operations performed. Sometimes, the description may be physical to a page but logical within a page. This approach is called *physiological*.

The only way to maintain the database page consistent at restart, is to flush all log records (that will be required to undo any uncommitted updates) with



After performing 3 actions, the transaction is partial rolled back by undoing 3 and 2, writing 3' and 2'. After this the transaction continues forward doing actions 4 and 5.

Figure 2.1: Savepoint and CLR

the page to disk. The *write-ahead log* (WAL) protocol ensures that a log flush is done before a page is put on stable storage. To maintain transaction durability, all log records of committed transactions must be made durable, helping restart redo the updates of those transactions. This rule is called *force-log-at-commit*.

In most systems undo is not the exact inverse of the redo operation, especially if logical logging is used. So, for each undo operation a *compensatory log record* (CLR) is generated. The CLR also invalidates the redo log that it undid. Therefore, the amount of logging performed during rollback, as well as the amount of undo during restart, is bounded (Refer *Figure 2.1*).

During execution of a transaction, if anything goes wrong, the application has to restart the transaction, giving up all that has been done. A *savepoint* can be established to remember the state of processing of the transaction, which an application now can rollback to, rather than abort the whole transaction (Refer *Figure 2.1*).

Checkpoint is a relatively recent persistent copy of the database state that can be used as the basis for restart processing. Checkpoints are used to speedup restart from failure. A *sharp checkpoint* puts the entire buffer of the database to stable storage in one synchronous operation. A *fuzzy checkpoint* writes the changed state to stable storage in parallel with normal operation. The pages are consistent individually, but on the whole they are not operation or transaction

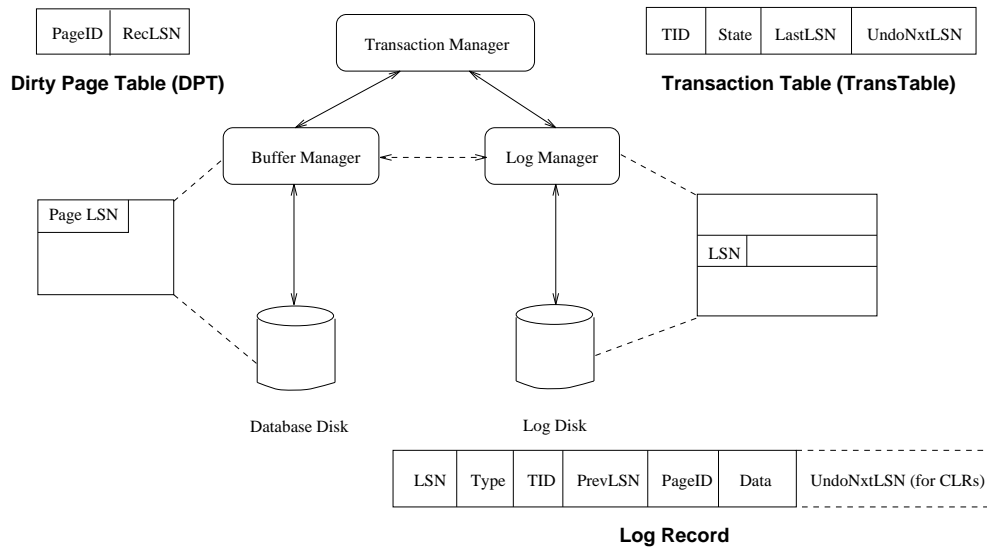


Figure 2.2: ARIES Model and Data Structures

consistent. A fuzzy checkpoint is made a sharp checkpoint by applying the log records generated during the checkpoint. This is an inherent part of restart recovery.

2.2 ARIES

ARIES – Algorithm for Recovery and Isolation Exploiting Semantics [MHL⁺92] is a recovery system which supports partial rollbacks, fine-granularity locking, WAL protocol and repeating history during restart before rolling back loser transactions.

ARIES supports fuzzy checkpointing. But the checkpoint does not flush dirty pages to stable storage. The modules of the system and the data structures are shown in *Figure 2.2*.

2.2.1 Updates

During normal execution of the DBS, transactions may be in forward processing, partial rollback or total rollback. If the granularity of locking is at record level, an update to a record in a page will be done as follows: (a) the record is locked, (b) the page is fixed in the buffer and latched in exclusive mode, (c) a log record added for the update, (d) the update is performed, (e) the LSN of the log record is placed in the page and transaction table entries, and (f) finally the page latch

is released and the page unfixes.

2.2.2 Total or Partial Rollbacks

The rollback routine of ARIES takes a savepoint upto which the rollback is performed for a given transaction. Rollback is performed assuming that all the locks of the transaction being rolled back are still held.

The log records are undone in reverse chronological order, and for each log record undone a CLR is written. When a CLR is written, its UndoNxtLSN field contains the PrevLSN of the log record whose undo caused the CLR to be written. Redo-only log records are ignored during rollback. The next log record to be undone for a non-CLR is determined by looking up the PrevLSN field. For a CLR, the UndoNxtLSN is used to determine the next record to be undone.

2.2.3 Restart Recovery

During restart recovery, ARIES scans the log from the first record of the last checkpoint, upto the end of log, performing *analysis*. The dirty pages and transactions that were logged in the checkpoint are brought upto date as of the end of log. Restart analysis outputs the transaction table (TransTable) containing the in-doubt transactions that were active in the system at the time of failure, the dirty pages table (RestartDPT) which is the list of potentially dirty pages in the buffer at the time of failure and the RedoLSN which is the start point on the log for the *redo* pass.

The RedoLSN and RestartDPT are supplied to the redo pass by the analysis phase. The redo pass scans the log from the RedoLSN point, and when a redoable log is encountered it checks if (a) the page is in the RestartDPT, (b) the log needs to be redone as per the RestartDPT, and (c) the page on disk does not contain the update. Only when the above conditions are satisfied is the redo performed.

Restart *undo* takes the TransTable and rolls back the loser transactions, in reverse chronological order, with a single sweep of the log. It writes CLRs for the log records being undone. The undo pass is conceptually the same as the total rollback explained earlier, with the only difference that all transactions of the TransTable are being rolled back.

2.2.4 Parallelism during Restart Recovery

In a parallel architecture, restart analysis can be performed independently at the respective nodes. According to [MHL⁺92], RestartDPT can help in initiating parallel asynchronous I/Os to read all pages with missing updates, so that they are available before the logs are encountered in the redo pass. Also, in-memory queues can be built of log records for a page or group of pages, and as the I/Os complete the logs can be applied in parallel by multiple processes. Undos of transactions can be distributed to various nodes, such that a transaction is undone only by one node, but at the same time the transaction undo load is equally distributed among the nodes.

2.3 Multi-Level Transactions and Recovery

A multi-level transaction (MLT) is made up of many sub-transactions. Each sub-transaction can do an early commit (called pre-commit) and observes ACID properties. If a parent transaction decides to rollback, it can semantically undo the pre-committed transactions by compensatory transactions. Multi-level recovery (MLR) copes with MLTs using a single log for all levels and thus presents a unified way of dealing with all levels [Lom92].

The essence of MLR is that for a level L_{i+1} , an operation OP_{i+1} at that level is undone by executing the inverse operation OP_{i+1}^{-1} called the compensatory operation. OP_{i+1}^{-1} returns the system to the state in which the effects of OP_{i+1} are not seen. An interrupted OP_{i+1} cannot be compensated by its OP_{i+1}^{-1} , but will be recovered by executing lower level compensatory operations.

Recovery of L_i undoes the OP_i s as seen in the log for incomplete OP_{i+1} s and writes compensatory log records. This ensures that the database becomes consistent as of level L_{i+1} . When all the OP_i s within an OP_{i+1} are compensated, the OP_{i+1} can be marked completed. Level L_0 is assumed to be atomic and the database is assumed to be stable at this level. The incomplete transactions of level L_1 can be recovered by compensating the constituent OP_0 s, hence making the level L_1 transactions consistent.

After a system crash, redo recovery is performed only at level L_0 . Then undo recovery is performed level by level. The level of the operation can be obtained from the log record. The simplest way to undo is to scan the log backwards multiple times, once for each level. The first scan is for undo of L_0 transactions that are part of incomplete L_1 transactions, after which the level is incremented by one. The scan at level L_i is done until no incomplete

transactions of the level L_{i+1} exists.

The main disadvantage is the amount of logging that is done in this scheme. Each level performs separate logging that contributes to the log overhead of even simple operations. Also the number of log scans during undo is greater than that under a flat transaction model. As suggested in [Lom92] the multiple scans can be reduced by maintaining in-memory queues of logs. This puts a lot of memory overhead for undo. Checkpointing logs will also be larger in this scheme as it will contain the transactions from several levels.

2.4 Recovery in a Shared-Disk Environment

In a SD environment a lot of new issues need to be addressed in recovery.

- Should the system use multiple logs (one per node) or should the logging be centralized?
- If the logging is done separately per node then how can the LSN be made monotonically increasing?
- How do we enforce the WAL protocol in a SD environment, where the dirty pages contain updates made at many nodes?
- How to perform checkpointing with multiple logs?
- How to perform restart recovery on a page, since log records of updates on a page may be scattered on different log disks? Is log merging to be performed?

2.4.1 Distributed Logging

Definitely, a centralized logging scheme is a bottleneck in the system. Every update or event to be logged, requires the node to send a message to the log server, which returns a LSN of the log record written. This increases the response time of the transaction. The node running the log server can be a candidate for failure, and to prevent the system from coming to a standstill without logging facility, a backup log server has to monitor and take over on primary failure. One of the visible advantages is the simplicity of the approach, such that almost all the issues listed in *Section 2.4* need not be considered.

2.4.2 LSN with Multiple Logs

The first intuition, for making the LSN monotonically increasing across logs at the nodes is to prefix a timestamp (assuming that the clocks at the nodes are synchronized) to the logical log address in order to obtain the LSN [MNP90]. But [MNP90] suggest that due to small size of the page LSN field the above strategy cannot be used. For example, the timestamp may require 6 bytes, but the page LSN field may allow only 4 bytes to be stored, leading to a truncated low-precision timestamp. As a result, updates to the same page by two nodes may have the same page LSN field value, leading to ambiguity and inconsistency.

If local clocks of the nodes are not synchronized, [MN94] propose *update sequence numbers* (USN) instead of page LSNs. During normal processing, the page USN is passed to the log manager at the local node. The log manager assigns a USN to the log record as the higher of the incremented values of the page USN and local USN counter (where, the local USN counter is the USN of the last log record written at the log manager). After the log record is written, the page USN and local LSN counter are updated to the log record USN. This scheme ensures that the log records written will have monotonically increasing USNs at the same node, and also for the same database page across nodes.

The LSN is no longer the logical log address in a multiple log system. So each page entry in the page table of the buffer manager remembers the physical address of the local log for the latest update on the page.

2.4.3 WAL Protocol in a SD Environment

In a SD environment, if a dirtied page is cached at one node, then another node wishing to access the page must get the current latest version of the page from the owner. This is managed by the cache coherency protocol in the system. In [MN91], the cache coherency protocol and recovery are tightly coupled. Cache coherency can be maintained in many ways like, requesting the owner to write the page to disk and reading from it, or asking the page via a message transfer.

The schemes for the transfer of page to the requesting node could be: *Simple*, *Medium*, *Fast* or *Super-Fast*.

Simple Scheme: In the simple scheme, the owner performs a disk I/O to write the page and after the I/O completes, a message is sent to the requester to read the page from disk. Due to the WAL protocol, there will be a implicit log force at the owner.

Medium Scheme: The medium scheme is the same as the simple scheme in principle, except that along with the page write to disk, the owner *also* ships the page directly to the requesting node.

Fast Scheme: This scheme differs from the medium scheme as follows: the owner of the page does not write the page to disk, when it is requested to relinquish ownership. Instead, it ships the page by a message and forces the log records to maintain the WAL protocol. The fast scheme provides better response time than the the previous schemes. In this scheme, dirty pages transferred between nodes may carry the updates of more than one node. Hence during recovery the logs of the nodes have to be merged to repeat history of potentially lost updates.

Super-Fast Scheme: The super-fast scheme does not require the log to be forced before shipping the page. In order to ensure WAL, when a dirty page is written to disk ultimately, this scheme requires tracking of the LSN values associated with a page on a per node basis. The page can be written to disk only after all updating nodes have forced their respective logs upto the LSNs being tracked.

2.4.4 Checkpointing and Restart Recovery

[MN91] handle page latch requests at two levels. The latch requests for pages are handled at the node by a *local lock manager* (LLM). The LLM forwards the request to a *global lock manager* (GLM) which keeps information of the oldest unapplied (to the disk version) logs on each page in the nodes.

Checkpointing needs to be done locally as the logs for the nodes are separate. As pages keep moving in and out of nodes there is a problem of pages being missed in the local checkpoints. So a *global-checkpoint* is performed of the GLM. This checkpoint is always used in the case of restart redo, both for system-wide or single-node failure. During restart redo from system failure or single-node failure, log scans are started from the minimum RecLSN values stored in the global-checkpoint and logs are merged based on the USN values.

2.5 Reduced Locking and Latching Using LSNs

Concurrency can be improved by avoiding lock requests that cause unnecessary waits and interference between transactions. Also fine-granularity locking is to

be done in such a way that it does not become too expensive for reading large quantities of data and for transactions which have low contention on affected pages.

[Moh90] introduces a novel method that uses the log record LSNs to reduce locking. The system maintains a *Commit_LSN* which is the minimum LSN of the starting point of transactions currently active in the system. The interpretation is that at least all updates logged before the *Commit_LSN* in the log have been committed. If the page LSN field is lesser than the *Commit_LSN* when the page is read, then all the data on the page is committed and locks need not be obtained by readers who require only degree 2 isolation.

In a SD environment, the system may be forced to get a global lock on the page to get the latest copy of the page before the *Commit_LSN* method can be applied. Also the *Commit_LSN* may be acquired by exchanging information between the various nodes in the system. The system still benefits from the elimination of the record level locks due to this method, which could save a lot of message passing between nodes.

A further optimization is to have a *Commit_Bit* assigned to each object in the page. The bit is set when the object is updated and bits are reset in the page if the *Commit_LSN* test is passed. With this bit, even if the *Commit_LSN* test fails, the reset state of the bit can be used to infer that the object is now in committed state.

2.6 Summary

In this chapter we have discussed some of the terms used in recovery and presented two commonly used recovery paradigms. Though MLR seemed attractive, its logging overhead was too expensive so we implemented the ARIES algorithm for multiple nodes. Some related work of recovery in shared-disk architectures has been discussed in this chapter. Many of our algorithms have been inspired by the ideas presented and some improvements have been proposed later.

Chapter 3

Software Architecture of *Brahmā*

Brahmā is a database system being built on the *Anupam* machine which has a SD architecture. Parts of *Brahmā* that had already been built, provided the interface to create objects on a Object Storage Manager. The detailed descriptions of the various components of the existing implementation can be obtained from [Des96, Gog96, Khi96]. However, for completeness, the components have been briefly discussed here. Also explained in this chapter is the coupling or interaction of existing modules of *Brahmā* and the new recovery modules.

3.1 Modules of *Brahmā*

The modules of *Brahmā* have been illustrated in *Figure 3.1*. A brief description of each is given below.

3.1.1 Operating System Layer (OS Layer)

The Anupam machine will be running UNIX on one of the nodes, that will control the activities of all the other nodes. Each of the other nodes will run a microkernel suited for *Brahmā*. Among other things this microkernel will provide means of communication between the nodes. The microkernel has a multi-threaded architecture, and therefore allows thread invocation.

As the microkernel is in development and the Anupam machine (hereafter also referred as system) is not yet available, it has been simulated by means of the OS Layer. The OS Layer was conceived and implemented for hiding the architectural details of the underlying OS on which the database engine

will eventually run. This makes our code highly portable; ready for any new machine, once the API of the OS Layer has been adapted for that machine.

Since the interface of the microkernel has been fixed as seen in the OS Layer, *Brahmā* is expected to work without any changes on Anupam.

3.1.2 Cache Manager (CM)

The Cache Manager runs on each node of the system. It is responsible for obtaining pages from disk or other nodes, whenever the upper layers request pages. Using the CM, users of *Brahmā* can read or write latch pages. The CM along with the Information Processors handles cache coherency. Pages are replaced using the local LRU policy.

CM maintains the *dirty page table* (DPT) which was introduced to hold the address of the oldest log of each cached page, that needs to be applied to the disk version of that page. This log address is called the ReLSN. The CM also interacts with the Checkpoint Manager during a checkpoint.

3.1.3 Information Processor (IP)

As *Brahmā* runs on a SD architecture, each page of the database is uniquely identified by a disk number and page number (hereafter referred as disk-page tuple). Since, latch information of a disk-page tuple needs to be visible globally to all the nodes, each node has an IP which maintains this information for a set of pages. All possible disk-page tuples are partitioned across the IPs in the system. The IP to which the disk-page tuple is assigned, is the sole decision making entity for that page. Each node wanting to latch a page or flush a page has to consult the corresponding IP.

The IP stores latch information as well as a list of nodes that hold the latest copy of the page in their local cache. This list is passed to a requesting node that needs the page, which then contacts one of the nodes on the list to obtain the latest copy. Before evicting a page, CM consults the appropriate IP to check if it needs to write the page to disk.

3.1.4 Storage Manager (SM)

The Storage Manager is responsible for allocating and freeing pages on disks. The SM also has the provision of allocating a set of pages called extents. To help the SM make decisions a *space map* is maintained on each disk. The space map consists of a bit per page to indicate whether the page is free or allocated.

3.1.5 Object Manager (OM)

Object Manager is the interface exposed to the users of *Brahmā*. The OM interface allows users to create, update, delete and retrieve untyped objects.

The OM consists of a set of other modules which perform various functions. New objects are assigned object identifiers (OID) to uniquely identify them and for this the `OidCounter` is referred by the OM. The `OidCounter` is persistent information which is maintained by the Header Manager (HM). Objects are searched on disks using the Logical-OID-to-Physical (LOOP) mapping. A typical map is a B⁺Tree which stores with every OID (the key) the disk-page information of the object's location. The map is updated when an object is moved, whereas object creation inserts a new entry and object deletion deletes the map entry.

The Object Storage Manager (OSM) is used by the OM to find space on a disk to insert an object. The OSM maintains a persistent synopsis of the free space on each page with two-bits in the *free space map*. The Cluster Storage Manager (CSM) allows the creation of a cluster of pages, which can then be used to store objects. The cluster is represented by an object called the cluster object, that maintains the free space information of pages in the cluster. Objects created can be hinted to belong to a cluster in the OM.

3.2 New Modules of *Brahmā*

New modules of *Brahmā* for transactions and recovery have been illustrated in *Figure 3.1* with dotted lines. A brief description of each is given below.

3.2.1 Transaction Manager (TM)

Users of *Brahmā* can use the Transaction Manager for initiating transactions. The Transaction Manager API provide calls to begin, abort and commit transactions. Transactions can be partially or fully rolled back. The list of transactions active at the node is maintained in a table called the `TransTable`.

3.2.2 Log Manager (LGM)

The primary functionality of Log Manager is to allow writing of logs. Typically checkpoint logs are big and cannot be put on one log page, so LGM provides the API to read and write split log records. LGM also has a log flush call which is needed for the WAL protocol.

3.3 Interaction between the Modules

The new modules of *Brahmā* interact with the existing modules to perform transaction management, checkpointing and recovery. The interaction has been numbered on *Figure 3.1* and enumerated below:

1. **CM to LGM:** (i) When the page is brought into the cache to record the RecLSN in the DPT, (ii) When the page is being flushed, logs are also flushed as per the WAL protocol.
2. **LGM to SM:** Allocating log extents.
3. **OM to LGM:** Logging updates.
4. **CKM to TM:** Checkpointing the TransTable.
5. **TM to RM:** Performing partial or complete rollback of transactions.
6. **RM to OM:** Performing physiological redos and logical undos during restart recovery and transaction rollback.
7. **RM to LGM:** Reading logs during rollback and restart recovery.
8. **CKM to LGM:** Writing the checkpoint logs.
9. **CKM to CM:** Checkpointing the DPT.

3.4 Summary

In this chapter, we have briefly discussed the existing software architecture and the newly implemented modules. The interaction between the modules has been illustrated. A more detailed description of the implemented modules is given in the following chapters.

Chapter 4

Overview of Recovery in *Brahmā*

The ARIES algorithm for recovery is explained briefly in *Section 2.2*. But, ARIES as described is for a single-node system. For *Brahmā*, we have added some features to ARIES. We will describe these changes and the recovery algorithms briefly in this chapter.

4.1 Normal Processing

A single log for all the nodes seems to be a bad choice. So we adopt the multiple log scheme; that is, each node maintains a separate log file. However, this means that updates to a page may be scattered across multiple logs. To tackle this issue, during redo, logs of the nodes are merged and ordered based on the USN stored with the logs.

4.1.1 Changes to DPT Entries

A dirty page containing the committed and/or un-committed updates of multiple nodes can be transferred from one node to another without being written to disk in the *Fast Scheme* [MN91]. In the *Fast Scheme*, a requested page is transferred between two nodes by the interconnection network, as a message, without being flushed to disk. Logs are flushed on the node from where the page is transferred. Whenever, a page latch is transferred, the node which gets the latch is responsible for writing the page to disk. It is also responsible for recovering the page in case of a crash.

A simple modification done to cache coherency is to pass the recovery point

with the page, when the page is transferred. In ARIES the DPT holds the RecLSN which is the recovery point. This is modified to RecUSN and RecLogAddr in *Brahmā*. So, each page carries with it the RecUSN and RecLogAddr information of each node in the system. The value represents the minimum redo log for the page on a node. As the page moves from one node to another, this information is also kept upto date at the nodes in their respective DPT entries.

For example, if a page $P1$ was fetched from disk and latched at Node 0 of a system with two processors, then the DPT entry for $P1$ will be $\{X, -1\}$ (where X is the current value of the USN counter at Node 0). Node 0 may give up the latch (while still holding a copy of $P1$ in its cache) letting Node 1 acquire it. At this point $P1$ is shipped, with the DPT entry for $P1$, from Node 0 to Node 1. Node 1 will add the given DPT entry and update it to reflect its local USN counter (say Y), making it $\{X, Y\}$. Now if the page is shipped back to Node 0 after Node 1 has finished with it, the DPT entry at Node 0 will also be updated to $\{X, Y\}$, even though a stale copy existed in the cache.

4.1.2 Cache Coherency

The WAL protocol and cache coherency followed is the *Fast Scheme* as proposed in [MN91]. All messages sent between two threads are guaranteed to be received in the same order. Otherwise there is no guarantee of message ordering between two different sets of threads.

It is also important to understand the cache coherency protocol. The cache coherency protocol explained in [Des96] is described briefly below.

Latching a Page: When the CM on a node requires to latch a page, it contacts the appropriate IP for the latch. The request is blocked, if another node is holding the latch in a conflicting mode, or the page is in transition¹. Else, IP returns the node (owner) from which the page can be obtained. The IP also marks the page entry to indicate page transition. If no owner is returned, the page can be read from disk.

If an owner is returned, the requesting CM requests the owner for a copy of the page. The owner then sends the page with the DPT entries to the requester by a message on the interconnection network. After receiving the page from the owner, the requesting CM sends an acknowledgment to the IP, which in turn updates the entry for the page. Also the requesting CM makes an entry for the

¹*transition*, means that the page is in movement from one node to another or from a node to disk

page in its DPT.

Disposing a Page: When the CM runs out of cache frames, it will evict a page that is not latched and then contacts the appropriate IP to check the status of the page currently in the cache frame that was chosen. If the page is in transition, IP blocks the request; otherwise the IP returns instructions to the CM after marking the page as in transition. The IP may direct the CM to discard the page (stale copy) or write the page (latest copy) to disk. On completion of the instruction, CM sends back an acknowledgment to the IP and removes the page entry from its DPT.

4.2 Checkpointing

A fuzzy checkpoint in ARIES records the state of the TransTable and the DPT. In a SD environment, this needs to be done at all the nodes in the system. So each node performs its individual checkpoint and writes the checkpoint log. The address of the checkpoint logs of all the nodes is stored in a master record on disk. The last step of the checkpoint is to write this master record atomically to disk.

Though the checkpoints are done independently at all the nodes, it needs to be ensured that pages don't escape the checkpoint by moving between nodes. One easy way would be to freeze both the DPT and the TransTable while the checkpoint is in progress affecting the response of normal transactions. Instead, the DPT and TransTable are not frozen. They are momentarily locked when an entry is read to be added to the checkpoint log. Also, a coordination protocol of messages is used between nodes, which will ensure that any page moving in or out of a node while a checkpoint is in progress is recorded.

4.3 Restart Recovery from System-Failure

Recovery from system-failure requires that the pages in the database are made consistent, and transactions which were active at the time of crash are rolled back at all the nodes. The generic algorithm to recover the system on restart is given below:

Perform Restart Analysis at the nodes
Synchronize the completion of analysis

Exchange RedoUSNs among the processors
 Perform Restart Redo from the merged log
 Synchronize the completion of redo
 Perform Restart Undo at the individual nodes
 Take a Checkpoint

Each node will perform analysis from the logs at the node. To obtain the minimum redo point for each node, analysis needs to be completed at all the nodes. When analysis is done at a node, the redo point seen by the node is sent to all other nodes. As soon as all redo points are obtained, logs are merged and redo is performed. This can be done by one node or the work can be distributed among the nodes. When all the pages are made consistent in the database, undo is performed at the individual nodes to rollback loser transactions.

4.4 Restart Recovery from Single-Node Failure

Failure of one node in the system should not bring the whole system to a standstill. That is, it should be possible for the other nodes to function normally even with a single node failure.

However, recovery from failure of a single node in the system, means that

- Updates made by the nodes on the pages that were latched on the failed node at the time of failure must be redone on the copies from disk.
- Updates by transactions of the failed node need to be undone.
- Data structures that may be distributed, like the IP, need to be rebuilt and restarted on the other nodes.

After the above has been achieved, the remaining system can function normally.

4.5 Summary

In this chapter, we have briefly presented the approach of performing restart recovery and checkpointing in *Brahmā*. This brief overview is followed by implementation details of the various modules and algorithms in the coming chapters. In this chapter we have also put forth the basic design changes that were done to move ARIES to a SD environment.

Chapter 5

Transaction Manager

In this chapter we discuss the implementation of the Transaction Manager(TM) along with some of its important components. The interface of the TM for users of *Brahmā* has been enumerated and a brief description of the functions has been given.

5.1 Components of the Transaction Manager

The transaction table (TransTable) is the main data structure of the Transaction Manager. It is a hash table that is used to maintain transaction information, and it allows quick access based on the transaction identifier or TID. TM consists of a transaction counter (TIDCounter) which is used to generate unique transaction identifiers for transactions on the node. TM also maintains the minimum starting USN of all the active transactions on the node, called CommitUSN.

5.1.1 Transaction Information

TransTable maintains information about transactions on the node in a `Transaction_t` structure. The structure holds the transaction identifier, last log written for the transaction, and the next log to be undone in case the transaction has to be rolled back.

USN of the first log written by the transaction is remembered in `StartUSN` and this is used for computing the `CommitUSN`. The `Status` field specifies whether the transaction is active or in the prepare state of two-phase commit.

```
TID_t      TID;
LogAddr_t  LastLogAddr;
LogAddr_t  UndoNextLogAddr;
```

```

USN_t      UndoNextUSN;
USN_t      StartUSN;
TwoByte_t  Status;
FourByte_t SavePtNum;
LogAddr_t* SavePoints;

```

The `Transaction_t` structure holds the list of savepoints for a given transaction. Partial rollbacks use the savepoint log address from the `Transaction_t` structure.

5.2 API of the Transaction Manager

The main interface of TM is to permit users of *Brahmā* to begin, commit and abort transactions. These functions are declared as follows:

```

Error_t TM_Begin(TID_t &TID);
Error_t TM_Rollback(TID_t TID, SavePoint_t spt = -1);
Error_t TM_Commit(TID_t TID);

```

Users can also obtain savepoints identifiers, as follows:

```

Error_t TM_SavePoint(TID_t TID, SavePoint_t &spt);

```

When this function is called, TM puts the last log address from `Transaction_t` into the `SavePoints` list and returns an identifier to the user. This identifier is then used in the rollback routine. The savepoint identifier as “-1” is used to indicate total rollback.

5.2.1 For Recovery

TM is also used during restart recovery. Restart analysis rebuilds the `TransTable` to the state that existed when failure occurred. Details of restart recovery can be found in *Chapter 10*. At the start of restart analysis, TM reads in the `TransTable` from the *end checkpoint* log data. While building the `TransTable`, restart analysis adds transactions encountered in the logs and removes them if it encounters the corresponding *end-transaction* logs.

TM has the API to provide data to the checkpointing routine of Checkpoint Manager. Checkpointing is done by latching the `TransTable` for the duration of obtaining a `Transaction_t` entry. After reading the structure, the latch is released while the log is being prepared. The `TransTable` is latched again before reading the next `Transaction_t` structure, minimizing the interference of checkpoints to

transactions and allowing greater concurrency. Currently the checkpointing of transactions does not include the savepoint data, but this can be easily added as an extension to the existing routines.

During the undo pass of restart recovery, TM aids in getting the transaction with the maximum undoable log address.

```
Error_t TM_GetMaxUndoLSN(Transaction_t** ppTrans);
```

This is done by checking the UndoNextUSN in the Transaction_t structures of the TransTable.

5.2.2 Computing CommitUSN

CommitUSN is calculated as the minimum StartUSN of the transactions in the TransTable. Whenever a transaction ends, the CommitUSN is likely to change if the ending transaction's StartUSN was the CommitUSN. Currently the CommitUSN is calculated at regular intervals preceded by a flush of logs. This may cause some extra locking; that is the CommitUSN may be old and transactions could be forced to acquire locks. However, this does not violate the correctness of the technique and reduces the overhead of calculating CommitUSN frequently.

5.3 Summary

In this chapter, we have seen the components of Transaction Manager. The API of TM for users of *Brahmā* has been briefly explained, along with parts involved in restart recovery. In the later chapters we will see the use of this API.

Chapter 6

Log Manager

The most important part of the recovery subsystem is the maintenance of logs. In this chapter we will discuss the Log Manager (LGM) and its various components. We will also describe the interface of the LGM as used by modules of *Brahmā*.

6.1 Logging

We have adopted the scheme of multiple logs; that is, a separate log file per node. It may not be a feasible option in the final running system to provide each node with a separate disk for logging, and so the logging mechanism has been adapted to writing logs on to a small set of disks, which could be lesser than the number of processors. Logging could also happen on the same disks on which data is written by the OM.

6.1.1 Log Address and Extents

Logs in the system are written in sets of pages called extents. Each log extent address, identified by `LogExtent_t`, consists of the disk number and the first page of the extent. The LGM on a node allocates equal-sized extents on disks through the SM. Each log can be addressed on stable storage by `LogAddr_t`, which is made up of:

```
Disk_t    DiskNo;  
Page_t    ExtPage;  
LogNum_t  LogNum;
```

The log address consists of the disk identifier, the first page of the extent and the log number within the extent. The log address is not monotonically

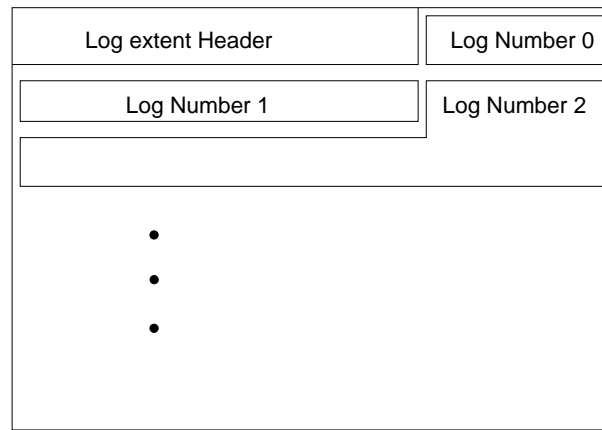


Figure 6.1: Sequential Log Extent

increasing, as the extents allocated on disk may not be with increasing page numbers. So a counter is prefixed to the address to make it increasing. This prefix is called the *update sequence number* or USN, and remembered as a `Local_USN_Counter` in the LGM.

Each log extent has a sequential architecture, as shown in *Figure 6.1* and consists of an header with two fields as shown below.

```
LogAddr_t    PrevAddr;
LogExtent_t  NextExtent;
```

`NextExtent` is used to form the forward chain of log extents, while `PrevAddr` stores the log address of the last log in the previous extent thereby forming the doubly linked list of extents.

6.1.2 Disk Log Header

The logs can be viewed as linked list of extents. To traverse the link list, we need to store the head, whereas to get to the end of the list with minimum I/Os we need to store the tail. So each node stores the first and last log extent information, at a well known location on disk for their respective logs, called the disk log header. The disk log header contains the information as shown below.

```
Proc_t       ProcNo;
LogExtent_t  FirstLogExtent;
LogExtent_t  LastLogExtent;
```

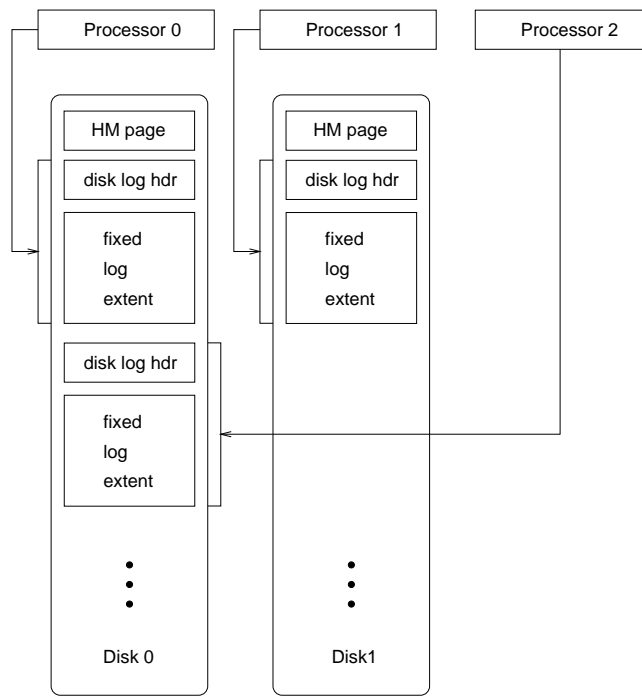


Figure 6.2: Disk Log Header and Preallocated Log Extents

At startup the last log extent is loaded so as to be able to append new logs. Whenever a new log extent is allocated the header information is updated on disk.

During extent allocation, SM updates disk information which needs to be logged. This cannot be done at system installation, as there will be no log extent to record a log extent allocation. A typical bootstrapping problem. To bypass this problem the disk log header is followed by a preallocated log extent for the processor. This is shown in *Figure 6.2*.

6.1.3 Initialization of Log Manager

When LGM is initialized, the disk log header is read from the assigned disk for the processor. This may not be initialized as in the case of installation. Then, it is initialized taking the preallocated log extent as the first log extent and the `Local_USN_Counter` is set to 0. If the disk log header is correct, then logging is continued on the last log extent as recorded in it. The `Local_USN_Counter` is initialized from the last log.

6.2 Types of Logs

Every log has an USN, a Type field to indicate the log type and a Length field of the log in the extent. In addition to the above common fields, each log has log specific information determined by the Type field.

LGM allows the typical logs of redo, redo-undo and CLRs as in ARIES. Other log types include end-transaction and checkpoint. Except the LOGTYPE_BEGINCHKPT log, which has no data, all the logs are illustrated in *Figure 6.3*. The common fields of USN, Type and Length have not been shown in the figure.

Speciality of the logs in *Brahmā* are presented below:

- The physical redo-only log, records the offset and data of the update applied to the page. LOGTYPE_REDO log is used where undo operation is not required.
- The LOGTYPE_EXTENTALLOC log is used by LGM to record allocation of extents for logging. When LGM requests log extents from the SM it prevents the SM from writing the log. After allocation this information is recorded in the existing log extent which has been filled up. So space for this special LOGTYPE_EXTENTALLOC log is always reserved in an extent.
- The physiological redo-undo log, LOGTYPE_PL_REDOUNDO, for an update remembers the redo and undo operation to be performed. The redo part is recorded as a physiological operation, along with the redo data. The undo part is kept as a logical operation. This type of logging greatly reduces the logging overhead for complex operations.
- LOGTYPE_PL_CLR is used to mark the compensatory actions of a physiological update operation and therefore, records only the redo opcode and redo data.
- In ARIES a nested-top-action is defined as a set of operations such that, if the whole set does not complete, the individual operations are undone. But, if the whole set completes the operations are not undone, indicated by a dummy CLR. In *Brahmā*, we have introduced a LOGTYPE_PL_NTA log inspired by this. This log differs in the fact that, the set of operations can be undone by a logical undo operation and the next undoable log address (which appears before the set of operations) is remembered in this log. This is further described in *Chapter 10*.

LOGTYPE_REDO

| | | | |
|------|------|--------|------|
| DISK | PAGE | OFFSET | DATA |
|------|------|--------|------|

LOGTYPE_EXTENTALLOC

| | |
|------|-------------------|
| DISK | EXTENT FIRST PAGE |
|------|-------------------|

LOGTYPE_PL_REDOUNDO

(INFO is {TYPE, SIZE, DATA})

| | | | | | |
|-----|------|------|---------|-----------|-----------|
| TID | DISK | PAGE | PREVLOG | REDO INFO | UNDO INFO |
|-----|------|------|---------|-----------|-----------|

LOGTYPE_PL_NTA

| | | | | | | |
|-----|------|------|---------|-------------|-----------|-----------|
| TID | DISK | PAGE | PREVLOG | UNDONEXTLOG | REDO INFO | UNDO INFO |
|-----|------|------|---------|-------------|-----------|-----------|

LOGTYPE_PL_CLR

| | | | | | |
|-----|---------|-------------|------|------|-----------|
| TID | PREVLOG | UNDONEXTLOG | DISK | PAGE | REDO INFO |
|-----|---------|-------------|------|------|-----------|

LOGTYPE_END

| |
|-----|
| TID |
|-----|

LOGTYPE_ENDCHKPT

| | |
|---------|------|
| PREVLOG | DATA |
|---------|------|

(Chain of split logs. LOGTYPE_BEGINCHKPT has no data)

LOGTYPE_OIDSET

| |
|------|
| DATA |
|------|

LOGTYPE_PAGEFETCH

| | |
|------|------|
| DISK | PAGE |
|------|------|

Figure 6.3: Log Types used in *Brahmā*

- In *Chapter 3* we have seen that the `OidCounter` is persistent information, therefore changes to it need to be logged. Since, updates to `OidCounter` are done in memory and the final value written to disk, special logging is done of these updates by writing `LOGTYPE_OIDSET` logs.
- Whenever a page is obtained via message transfer, the requesting node writes a special `LOGTYPE_PAGEFETCH` log, if the requesting node hosts the IP for the page. This log has to be written for correctness of restart recovery on single-node failure. The use of this log is explained in *Section 11.4.2*.

6.3 Log Cache

LGM has a local cache for reading from and writing into log extents. This cache is made up of frames which hold log extents. Each frame has the log extent address on disk, a buffer for the log extent, a fix count and a lock. The fields are shown below:

```

LogExtent_t      Extent;
TwoByte_t        FixCount;
OneByte_t*       Buf;
pthread_mutex_t  MutexBuf;
LogNum_t          NumberOfLogs;
LogSize_t         FreeSpacePtr;

```

The lock is used to latch the log extent while reading from or writing into the extent. `NumberOfLogs` specifies the number of logs currently in the extent, whereas `FreeSpacePtr` gives the point from which the next log can be written in the extent. A log extent does not store any information about the number of logs in it. Therefore, to calculate the last log within the extent a special marker is to be used. Also using this marker, the USN from the last log in the extent can be determined. If no log exists in the extent, the marker is followed by the last USN.

The log cache is made up of two parts, a frame permanently allocated for writing logs (write-frame) and a set of frames for reading logs (read-frames). Transactions reading logs will first fix the log extent into one of the read-frames, and while reading, latch the frame for a short duration to get a consistent view. This is done even if the log extent to be read is the same as the extent in the write-frame.

Fixing a log extent will get its latest version into a read-frame. If the latest copy already exists in one of the read-frames then its fix count is incremented. Otherwise, the latest copy could be fetched from the log disk or obtained from the write-frame.

The algorithm for fixing log extents used by readers is given below in pseudo-code. It returns the read-frame which has been latched and fixed in the log cache.

Function FixNLatchLogExtent (LogExtent P)

Uses: Hash Table H , read-frames R , write-frame W

Returns: read-frame r where the extent is loaded. This frame is latched and fixed.

begin

Lock the Hash Table H

Check in which read-frame of R is extent P loaded. This returns an index i .

if i is valid **then**

 Latch the read-frame $R[i]$

 Latch the write-frame W

 Release lock on H

if Extent in $R[i] =$ Extent in W **then**

 Copy buffer from W to $R[i]$

end if

 UnLatch W

else

 Select a read-frame i for P . This latches $R[i]$.

 Latch the write-frame W

 Release lock on H

if $P =$ Extent in W **then**

 Copy buffer from W to $R[i]$

 UnLatch W

else

 UnLatch W

 Read the log extent P from disk into the read-frame $R[i]$

end if

end if

Increment Fix count of read-frame $R[i]$

$r \leftarrow R[i]$

```

    return r
end

```

Whenever the log extent needs to be loaded on to a new frame then a frame has to be selected. This could be a frame with fix count of 0. If no such frame exists then a new frame is allocated. The pseudo-code for selecting a frame is shown below. While searching for an empty frame it is assumed that the whole hash table of read-frames are locked as in the previous algorithm.

Function SelectFrame (LogExtent P)

Uses: Hash Table H , read-frames R

Returns: Index i of the read-frame allocated

begin

Search in R for a frame i with fix count 0

if i is not valid **then**

 Latch all read-frames R for resizing the log cache

 Resize the log cache and allocate a frame

 UnLatch all but the newly added frame

$i \leftarrow$ Index of the new frame

else if an extent is already loaded in the selected read-frame $R[i]$ **then**

 Delete the extent entry from H

end if

Insert extent P into the hash table H with an index i

return i

end

6.4 Writing Logs

All the modules of *Brahmā* use the LGM to write log records for updates, events and checkpoint information. The main API of LGM for writing log records is as shown below.

```

Error_t LGM_WriteLog(OneByte_t type, OneByte_t* logData,
                    FourByte_t size, USN_t &USN,
                    LogAddr_t &addr);

```

This function accepts the log type and data, and returns the USN and address of the log written. The USN is read from the page (if log is for an update) and passed to this routine. The log USN is set as per the description in *Section 2.4.2*.

The write-frame is used for writing logs into the last log extent. This frame is exclusively latched while writing a log, since many transactions use the same LGM to perform logging. The `NumberOfLogs` and `FreeSpacePtr` are appropriately updated in the write-frame when a log is written. The pseudo-code of writing a log is given below.

Function WriteLog (Log Type T , Log Data D)

Uses: Hash Table H , frames W and R , Local_USN_Counter

Returns: USN of log, Addr of log

begin

Latch the write-frame W

while true do

Check if there is space for this log and the LOGTYPE_EXTENTALLOC log

if space exists then

USN \leftarrow max(USN, Local_USN_Counter) + 1

Write the log in W with USN

Addr \leftarrow { W .Extent, W .NumberOfLogs}

Set the Local_USN_Counter and page USN field to USN

Update W .NumberOfLogs, W .FreeSpacePtr

UnLatch the write-frame W

return log USN and log Addr

else

Remember the log extent of W in a variable, say $X1$

UnLatch the write-frame W

Lock the hash table H

Search if W 's extent is loaded in any read frame i .

if i is valid then

Set Flag F

Latch the read-frame $R[i]$

end if

Latch the write-frame W

Release lock on H

if W .Extent \neq $X1$ then

```

    if  $F$  is set then
        UnLatch read-frame  $R[i]$ 
    end if
else
    Allocate a new log extent from the SM (does not log)
    Write a LOGTYPE_EXTENTALLOC log in  $W$ 
    Update the header NextExtent to set log extent chain
    if  $F$  is set then
        Update the read-frame  $R[i]$  from  $W$ 
        UnLatch the read-frame  $R[i]$ 
    end if
    Flush log extent  $X1$  to disk
    Set new extent in the write-frame  $W$ 
    Flush new log extent header to disk
    Update the last log extent pointer in the log header page
end if
end if
end while
end

```

Even though the last log extent is updated in two places, which is not atomic, it can be verified at startup. If the last log extent contains a non-zero NextExtent, it means the information in the log header page is invalid and can be rectified. This is done at startup, during initialization of the LGM.

6.4.1 Splitting Big Logs

Sometimes a log may be too big to fit in a single log extent. A typical example is a checkpoint log record. Under such cases the user may use the following routines to write split logs and read split logs. Split logs are stored in a reverse chain as shown in *Figure 6.4*. While reading the split log, the last log in the chain is read first and the log data is reconstructed by chaining backwards.

```

Error_t LGM_WriteSplitLog(OneByte_t type,
                          OneByte_t* logData,
                          FourByte_t size,
                          USN_t &USN, LogAddr_t &addr);
Error_t LGM_GetMergedLogData(LogAddr_t lastaddr,

```

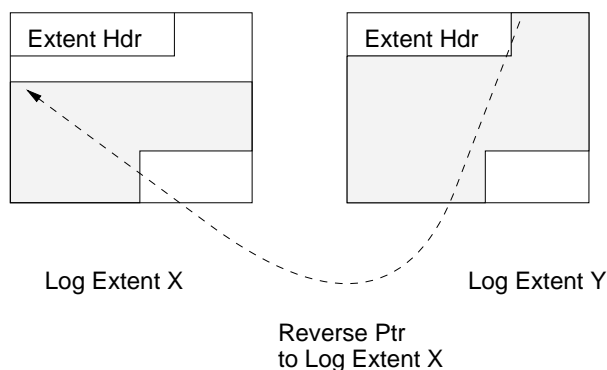


Figure 6.4: Split Logs put in Log Extents

```
FourByte_t &logsize,
OneByte_t** logdata);
```

6.5 Log Flush

The LGM allows logs to be flushed as per the WAL protocol or on commit of transactions. The calls for flushing logs is shown below.

```
Error_t LGM_FlushLog(USN_t USN);
```

The flushing of a log extent means writing multiple pages to disk, which leads to two problems. The first is of making the log writes atomic. Writes in the OS can be assumed atomic only at page level. But the extent is a set of pages. So we flush the pages of the extent in the reverse order of pages. Thereby, ensuring that the special marker in the extent is always on disk before the logs.

The second problem is the number of I/Os on log flushes. If the whole write-frame is to be written on each log flush, it would cause a lot of I/Os. To reduce I/Os we maintain two pieces of information. One is the last flushed log address and USN. The other is a flush bit per page of the log extent. Every time a completely filled page of the extent is written to disk, the bit is set indicating that the next flush need not rewrite this page.

6.6 Log Iterator

Though writing logs in *Brahmā* is an important function, reading logs while performing rollback and restart recovery is also needed. LGM provides the interface to iterate through the logs either sequentially or in a random fashion. The random backward scan is used during partial or total rollbacks.

The log iterator is initialized by specifying a valid log address and calling,

```
Error_t LGM_IterInit(LogAddr_t p, LogIterState_t** state);
```

If the log address is the next log address that will be written then the call returns a end-of-log status. The log extent containing the log is fixed in a log cache read-frame, and the iterator state is returned. The state contains the read-frame identifier of the cache where the log extent is loaded and other information about the log such as data, address, USN, type and size. This is shown below,

```
LogFrameId_t FrameId;
LogAddr_t     LogAddr;
OneByte_t*   LogData;
USN_t        USN;
LogType_t    Type;
LogSize_t    Size;
```

The log data is given by an in-memory pointer in the fixed read-frame. This allows multiple readers to use the same extent in the read-frame to read logs, assuming that the extent buffer is contiguous and allowing logs spanning page boundaries.

For scanning logs the API provided is:

```
Error_t LGM_IterNext(LogIterState_t* state);
Error_t LGM_IterPrev(LogIterState_t* state);
Error_t LGM_IterGoto(LogIterState_t* state, LogAddr_t p);
```

Since there is considerable similarity in the implementation of the above APIs, only LGM_IterNext pseudo-code is presented below.

Function IteratorNext (LogIteratorState *S*)

Uses: read-frames *R*

Returns: New state in *S* or End-Of-Log if no next log exists

begin


```

        OneByte_t* pageBuf, OpCode_t redoType,
        Offset_t redoSize, OneByte_t* redoData,
        OpCode_t undoType, Offset_t undoSize,
        OneByte_t* undoData);
Error_t LGM_Update_pLNTA(Transaction_t* pTrans,
        Disk_t disk, Page_t page,
        LogAddr_t UndoNext, OneByte_t* pageBuf,
        OpCode_t redoType, Offset_t redoSize,
        OneByte_t* redoData, OpCode_t undoType,
        Offset_t undoSize, OneByte_t* undoData);
Error_t LGM_Update_pCLR(Transaction_t* pTrans,
        Disk_t disk, Page_t page,
        LogAddr_t UndoNext, OneByte_t* pageBuf,
        OpCode_t redoType, Offset_t redoSize,
        OneByte_t* redoData);

```

6.8 OverWriteUSN

As a log extent gets full, a new log extent is allocated for writing logs. But in most systems, logs are implemented in a circular buffer. That is when a log extent gets full, the first log extent is reused if possible. LGM has an OverWriteUSN field to indicate that log extents before this value can be reused.

Logs before the CommitUSN of TM will never be used for rollback of transactions. This is because the CommitUSN is the minimum starting USN of transactions active in the system. The DPT stores the recovery point for pages in the local cache. Logs before the minimum RecUSN in the DPT will never be used to bring pages upto date with the changes done.

The two concepts have been combined to determine the OverWriteUSN. While performing a checkpoint when DPT is scanned, the minimum RecUSN is returned to the LGM. Occasionally, TM calculates the CommitUSN which is passed to the LGM. The minimum of the two is stored as the OverWriteUSN.

Though the current implementation does not reuse log extents, code can be modified easily in the LGM_WriteLog function, to select the first log extent if the criterion is satisfied.

6.9 Summary

In this chapter we have described the components of the Log Manager and the algorithms used to write logs. In the next few chapters we will see the examples of logging and the use of LGM for recovery.

Chapter 7

Examples of Logging in the Modules of *Brahmā*

In *Chapter 6*, we have seen the Log Manager API for writing logs. Each node in the system runs the OM and other components of *Brahmā*, which perform updates on the shared disk database. But, each node writes separate logs. In this chapter we will see some examples of logging incorporated into the components of *Brahmā*.

Typically the same routine is used in case of forward, redo and undo processing. The processing is indicated to the routine by passing a flag. If the flag is set to undo processing, a UndoNext log address is passed which is used in the CLR's UndoNextLog field. During redo processing, the operation is repeated on the page but no log is generated.

7.1 In the Object Manager

Here we describe the algorithms for the creation, deletion and updation of objects along with extensions for recovery. As part of recovery we added logging calls to the operations, and these have been underlined in the algorithms. Logging also modifies the page USN field which is not mentioned in the algorithms.

7.1.1 Object Creation

Object creation is done by getting an unique OID from HM. Then, the OSM is requested to find a page that can fit the object, failing which a new page is allocated. Each object's location can be obtained from the Logical-OID-To-Physical (LOOP) map. This map is updated when an object is created. Finally,

the object data is inserted into the page.

The logical undo of object creation is object deletion. A NTA log storing this information is written if all operations are completed successfully. In case this routine is used for the logical undo of a delete object, a CLR is written instead of the NTA.

Function CreateObject (Object Data, Transaction Id TID, UndoNext log address, Processing flag PFlag. If the PFlag is “Undo” then OID is supplied. If the PFlag is set to “Redo” the {disk, page} is also passed.)

Uses:

Returns: OID of the object

begin

```

if PFlag  $\neq$  “Redo” then
  if PFlag  $\neq$  “Undo” then
    Get new OID
  end if
  Find space for object using OSM
  if space is not found then
    Allocate a new page from the SM
  end if
  Insert LOOP map entry {disk, page} for OID
  WriteLatch {disk, page}
  Update the Free Space Info for the page
  if PFlag  $\neq$  “Undo” then
    Put NTA log with delete of object
    with OID as undo and redo by inserting object
    into same page
  else
    Put a CLR log with the given UndoNext
  end if
else
  WriteLatch {disk, page}
end if
  Add new slot entry in the page
  Insert object into the page

```

```

Update data page header
UnLatch {disk, page}
return OID
end

```

7.1.2 Object Deletion

For object deletion, user supplies the OID of the object to be deleted. The routine latches the page where the object exists, and updates the slot entry as well as the header information. All this is logged by a NTA log to necessitate a logical undo (by an object insert). The delete operation finishes off by removing the LOOP map entry for the object. This routine is used by – the user to delete an object, delete exercised due to the logical undo of an insert operation, and in restart redo to repeat an object deletion from a page.

Function DeleteObject (Object OID, Transaction Id TID, UndoNext log address, Processing flag PFlag. If the PFlag is set to “Redo” the {disk, page} is also passed.)

Uses:

Returns:

```

begin
  if PFlag ≠ “Redo” then
    Get LOOP mapping {disk, page} of object OID
    WriteLatch {disk, page}
    Copy object data into buffer A
    if PFlag ≠ “Undo” then
      Put NTA log with, undo operation as
      insertion of object given OID and data A,
      and delete from the same page as redo operation
    else
      Put a CLR log with the given UndoNext
      Also put the redo data in the log
    end if
  else
    WriteLatch {disk, page}
  end if
end

```

```

Update slot entry of object on the page
Update data page header
Update the Free Space Info for the page
UnLatch {disk, page}
Delete LOOP map info of the {disk, page}
return
end

```

7.1.3 Object Updation

Updating an object is a tricky operation as its size may change. If the object size increases, then it may be moved in the same page or to a different page. We summarize the various cases of an object update here.

1. The object size remains the same but its content changes: In this case the object is replaced at the same location on the page.
2. The object size decreases by the updation: The object is updated at the same location on the page and the free space information in the header is changed. Later, a compaction operation may use up the free space at the end of the object.
3. The object size increases: One of the two cases may exist,
 - (a) The object fits in the same page and so it may be moved to another location on the page.
 - (b) The object does not fit in the same page. So, the object is removed from the page and inserted into another page that has enough space for it. This case we will discuss in the pseudo-code below.

Function UpdateObject (Object OID, Transaction Id TID, UndoNext
log address, Processing flag PFlag)

Uses:

Returns:

```

begin
  if PFlag = "Normal" then
    UndoNext  $\leftarrow$  LastLog of the transaction of TID
  end if

```

```

TempAddr ← LastLog of the transaction of TID
Get LOOP mapping {disk, page} of object OID
WriteLatch {disk, page}
Copy object data into buffer A
Remove object from the page (as in Delete) giving the PFlag and TempAddr
UnLatch {disk, page}
Insert updated object in a new page (as in Create) passing PFlag and Un-
doNext
return
end

```

7.2 In the Object Storage Manager

Here the most important routine is the updation of the free space information of the page. This routine is called by the OM and CSM to keep synopsis information about the page contents.

The updation of free space in the OSM is controlled by three flags. Whenever the free space information is updated and the page becomes empty, OSM deallocates (or frees) the page. The normal freeing of a page if empty, is controlled by the FreeFlag. During undo processing the routine is passed “Undo” in the PFlag as in all cases. This flag will ensure that free space updation is completed by writing a CLR and not a redo-undo log. The UndoFreeFlag is used to indicate in the redo-undo log of normal processing as to whether an empty page is to be freed while undo of the operation is done. Thus, the UndoFreeFlag will become the FreeFlag for the same routine during undo.

Below, we summarize the uses of the OSM to update the free space information:

1. **Create Object:** While creating an object a page may be allocated and the free space information updated. If the object creation does not complete, then the undo of free space information does not free the page as this will be undone by the SM. (FreeFlag = X, PFlag = “Normal”, UndoFreeFlag = 0)
2. **Delete Object:** After deleting an object from the page, the update free space information is performed. This may free the page. Since the freeing of the page is done after object deletion, the page delete need not be

undone. This is because during undo the object may be recreated on another page. So a CLR can be written for the free space information update even during forward processing for an object deletion. (FreeFlag = 1, PFlag = "Undo", UndoFreeFlag = X)

3. **Update Object:** While updating an object, the free space information update is to be undone by the same operation. This will cause no problems unless the object is removed from the page, in which case the page may get freed (The above scenario is shown in *Section 7.1.3*). For all other cases when the object is updated on the same page there is no danger of the page being freed in the update operation. (FreeFlag = X, PFlag = "Normal", UndoFreeFlag = 1)

Below, we describe the pseudo-code of the free space updation routine of the OSM.

Function UpdateFreeSpaceInfo (Transaction Id TID, OM page P , Free space page M , UndoNext log address, Processing flag PFlag, FreeFlag, UndoFreeFlag)

Uses:

Returns:

begin

WriteLatch the free space page M

Set the new slot value of page P

if PFlag \neq "Undo" **then**

Put a physiological redo-undo log for the slot value update

Also put the UndoFreeFlag in the log

else

Put a CLR log with given UndoNext

end if

UnLatch the free space page M

if FreeFlag and the page P is empty **then**

Deallocate page P in the SM

end if

return

end

7.3 Summary

In this chapter, we saw some examples of logging in the modules of *Brahmā*. Since the design of the OM in *Brahmā* was good, logging was added with minimal changes. Wherever possible the existing routines have been used to perform the redo and undo operations also. Undo and redo processing are indicated by passing a flag called PFlag to the relevant routines.

Chapter 8

Checkpointing Protocol and Checkpoint Manager

Checkpointing is essential in reducing the amount of work done during recovery. In this chapter we discuss the issues in checkpointing and present the checkpointing protocol.

8.1 Checkpoint Coordinator

A fuzzy checkpoint on a single-node system captures the state of the system along with normal operation. In *Brahmā*, a checkpoint is started synchronously for all the nodes by a checkpoint coordinator (CKMCOORD). It is usually started as the number of logs written reaches a threshold or at the end of restart recovery.

In each message during the checkpointing protocol, the nodes also exchange the checkpoint number which is called the Counter. This Counter is incremented by the CKMCOORD at the start of each checkpoint. As soon as CKMCOORD decides to start a checkpoint, it sets the Counter value and sends a `MESG_BC` message to start the checkpoint at each the node. This message is sent to the Cache Manager (CM) of the nodes.

Having started the checkpoint, the CKMCOORD waits for the completion of the checkpoint. This is indicated by to it by a `MESG_EC` message from each node. The `MESG_EC` message contains the Counter which should match the current checkpoint counter, failing which the message is ignored. The message also contains addresses of the *begin-checkpoint* and *end-checkpoint* logs written on the node, and the minimum RedoUSNs for each node as found by the checkpointing node. The RedoUSNs are collated, merged and then the RedoUSN of

each node is used to set the `OverWriteUSN` value at that node.

If a node fails in the system, the CKMCOORD may wait indefinitely for a `MESG_EC` message. To prevent this, on a node failure CKMCOORD receives a `MESG_NODEFAILURE` message from the node that detected failure. This resets the checkpointing status in the `InProgress` flag. Only when the node is recovered, is the checkpointing restarted indicated to CKMCOORD by a `MESG_STARTCHKPT`. The number of node failures is also kept in mind when restarting the checkpoint.

The CKMCOORD is also responsible for reading and writing the master checkpoint record. The master checkpoint record is read at restart recovery from system-failure or single-node failure. At the end of a checkpoint, CKMCOORD writes the master record, containing checkpoint addresses, atomically to disk.

8.2 Checkpoint Manager (CKM)

In *Brahmā* we have the Checkpoint Manager running on each node in the system to perform fuzzy checkpointing. Checkpoints are performed at the node with the help of the CM.

When CKM receives the `MESG_BC`, it begins the checkpoint as a separate thread at the node. The working of the thread is shown below.

Function CheckpointThread (Checkpoint Counter)

Uses:

Returns:

begin

Clear all data structures

Set `InProgress` flag

`BeginChkpt` \Leftarrow Write a begin checkpoint log

`Cntr` \Leftarrow Get the `OidCounter` from HM

`Dptbuf` \Leftarrow Log the DPT in a buffer

`TTbuf` \Leftarrow Log the `TransTable` in a buffer

Reset `InProgress` flag

`EndChkpt` \Leftarrow Write a split log with `{Cntr, Dptbuf, TTbuf}`

Send a `MESG_EC` message on completion to CKMCOORD with `{BeginChkpt, EndChkpt}`

end

The ongoing checkpointing is indicated by a `InProgress` flag. The *begin-checkpoint* log is written and the address is remembered to be sent finally to the CKMCOORD. DPT, `OidCounter` and `TransTable` are written to buffers, to be included in the *end-checkpoint* log

The `OidCounter` is information that needs to be maintained upto date on disk. Every new object is assigned an unique OID from this counter. Since every object creation will cause an extra I/O, the counter is maintained transiently and on shutdown of the database its value is written to disk. On restart recovery it is imperative that the counter's value be reconstructed. This is done by reading the checkpoint log and the `LOGTYPE_OLDSET` logs.

While reading the whole DPT, it is not locked as this will reduce concurrency. Checkpointing is done by latching the DPT for the duration of obtaining an entry and releasing the latch when the log is prepared. The DPT is latched again before reading the next entry, minimizing the interference of checkpoints to transactions and allowing greater concurrency.

As the page entries are read and checkpointed, CKM also adds them to a separate list called the `DoneList`. To prevent pages from escaping the checkpoint, whenever a DPT entry is removed or added by the CM, the `DoneList` is referred, and if the item is not found it is added to the checkpoint. During the checkpoint at any time if CKM changes its `Counter`, the checkpointing thread stops and clears all its data structures.

8.2.1 Issues in Checkpointing

We would like to ensure that a dirty page in the system does not escape checkpointing at all nodes. As we have seen in the above algorithm, the `InProgress` flag is set to indicate the checkpoint to the CM, and note changes to the DPT. Below we consider the cases that affect the DPT and their effect on an ongoing checkpoint.

Case 1: *DPT entry exists when the `InProgress` flag is set*

For the case when the DPT entry at a site (which existed before the flag was set) is removed after the flag is set (or even reset), it is guaranteed that the entry will exist in this checkpoint at that site. This is because, either the CM (if the entry is being removed) or CKM will add the page to checkpoint log record.

Case 2: *DPT entry is removed before the `InProgress` flag is set*

The case of concern occurs when the page is being shipped around such

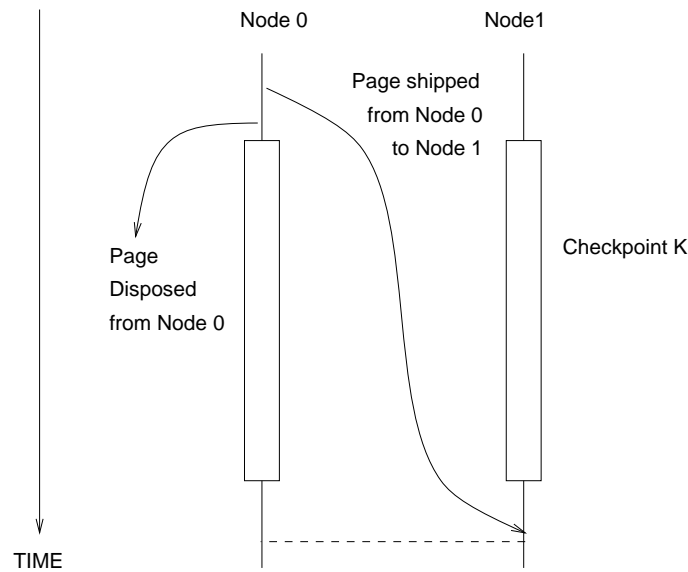


Figure 8.1: Page reaching a node after checkpoint at both nodes

that the page entry escapes the checkpoint at every node. This can only happen if the page entry from the DPT is removed before the flag is set, such that it does not exist in the DPT of any node during the checkpoint.

Case 2.1: *DPT entry is removed as page is flushed to disk*

If the page is being disposed to disk before the checkpoint, its entry is removed and the checkpoint may not contain the page entry at any node. This is fine as the page is latest on disk and need not be considered for restart redo.

Case 2.2: *DPT entry is removed as page has been shipped*

The other case is of the page being disposed from cache as it is not the latest copy. This may may lead to its entry being missed in the checkpoint at all nodes.

Case 2.2.1: Let us take the case as shown in *Figure 8.1*. Consider a page P which has been sent via message from Node 0 to Node 1. Let us assume a checkpoint K is in progress at all nodes, the duration of which is shown in the figure by a rectangular box.

As per the cache coherency protocol, the IP will not allow the page to be disposed from the cache of Node 0 until Node 1 acknowledges receiving the page. Therefore the case as shown in *Figure 8.1* cannot occur and the

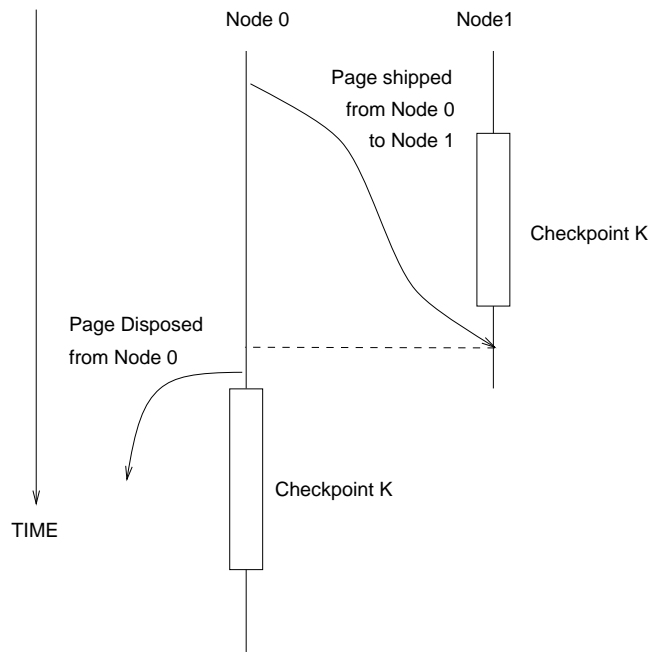


Figure 8.2: Page is shipped and disposed before the sender's checkpoint but reaches the requester after its checkpoint

page P will be checkpointed at Node 0.

Claim: A page in transition is guaranteed to exist in the DPT of at least one node.

Case 2.2.2: In the other case, the page P is sent and disposed from Node 0 before the checkpoint K and received by Node 1 after the same checkpoint K . This is shown in *Figure 8.2*. In this scenario the page will not be checkpointed at Node 0 as well as Node 1. The main cause of the problem is the non-deterministic arrival of `MESG_BC` messages to the nodes. Some nodes may get the message before others and finish the checkpoint before others have begun.

To solve the problem in **Case 2.2.2**, we use a coordination protocol. As soon as the checkpoint is started, coordination messages (`MESG_CS`) are sent from each node to the CM of all the nodes. Before the DPT is scanned at a node for checkpoint K , it is ensured that all the `MESG_CS` coordination messages are received by the node. The protocol is also shown diagrammatically in *Figure 8.3*.

Function CheckpointThread (Checkpoint Counter)*Uses:**Returns:***begin**

Clear all data structures

Set InProgress flag

for $i \leftarrow 1$ to NUMPROCESSORS **do**Send MESH_CS to Node i **end for**BeginChkpt \leftarrow Write a begin checkpoint logCntr \leftarrow Get the OidCounter from HM**for** $i \leftarrow 1$ to NUMPROCESSORS **do**Wait for MESH_CS from Node i **end for**Dptbuf \leftarrow Log the DPT in a bufferTTbuf \leftarrow Log the TransTable in a buffer

Reset InProgress flag

EndChkpt \leftarrow Write a split log with {Cntr, Dptbuf, TTbuf}

Send a MESH_EC message on completion to CKMCOORD with {BeginChkpt, EndChkpt}

end

One difference from the algorithm stated in *Section 8.2* is that the flag may be set at a node when it receives a MESH_BC or the coordination message MESH_CS. This is because there need not be a strict ordering of the MESH_BC reaching a Node i , before a Node j receiving the MESH_BC message and sending the MESH_CS message to Node i . All coordination messages also carry the Counter so that old messages can be discarded.

To prove our algorithm to be correct we need to compare it with the result of the simple checkpointing algorithm. In the simple algorithm, at some time t_c , we freeze the DPT entries of all nodes and ensure that there is no page in transition. At this point the simple algorithm puts all the DPT entries to the logs and finally synchronously unlocks the DPTs.

For ease we can rewrite our algorithm in five steps, and subscript each message by the nodes sending and receiving the message. So at Node i checkpoint K is done as follows: (1) Receive BC_i or CS_{ji} (set the InProgress flag), (2) Send

all CS_{ij} , (3) Wait for all CS_{ji} , (4) Read the DPT to perform checkpoint (finally resetting `lnProgress` flag), and (5) Send EC_i to coordinator.

The time instant t_c is to be chosen as the time when every node has set its `lnProgress` flag. That is the point when every node has received a `MESG_BC` or a first `MESG_CS` message for the checkpoint K .

Theorem 8.2.1 *The DPT entries of all the nodes at the time instant t_c are guaranteed to be in checkpoint K , where t_c is the time instant when every node has set its `lnProgress` flag for checkpoint K .* \square

Proof: As soon as the `lnProgress` flag is set at a Node i , it is guaranteed that changes to the DPT are being recorded in checkpoint K at that node. So, at time instant t_c all nodes are recording changes to the DPT.

Each node will start reading its DPT only when all coordination messages are received from the other nodes as shown in Step (3). This point is guaranteed to be after time t_c , because coordination messages are sent to other nodes only after setting the `lnProgress` flag. Therefore to complete Step (3) means each node has set its `lnProgress` flag.

The set of pages that have been disposed from Node i since the `lnProgress` flag was set, along with the set of pages recorded from the DPT at Node i , will be the superset of the DPT entries of Node i at time t_c .

Hence the theorem follows. \square

8.2.2 Improving the Algorithm

Though the above algorithm has been implemented it can be further improved. Clearly, resetting the `lnProgress` at a node cannot be allowed before the setting of the flag at another node.

So rather than blocking the checkpoint thread from reading the DPT, it can be blocked from writing the *end-checkpoint* log until all the coordination `MESG_CS` messages are received. Till then, even if the DPT scan is over, the pages that come into the DPT are added to the `Dptbuf`. The above proof can be easily changed to accommodate this modification.

8.3 Summary

In this chapter we have seen the checkpointing protocol followed in *Brahmā*. Fuzzy checkpointing is started synchronously at all the nodes and then the

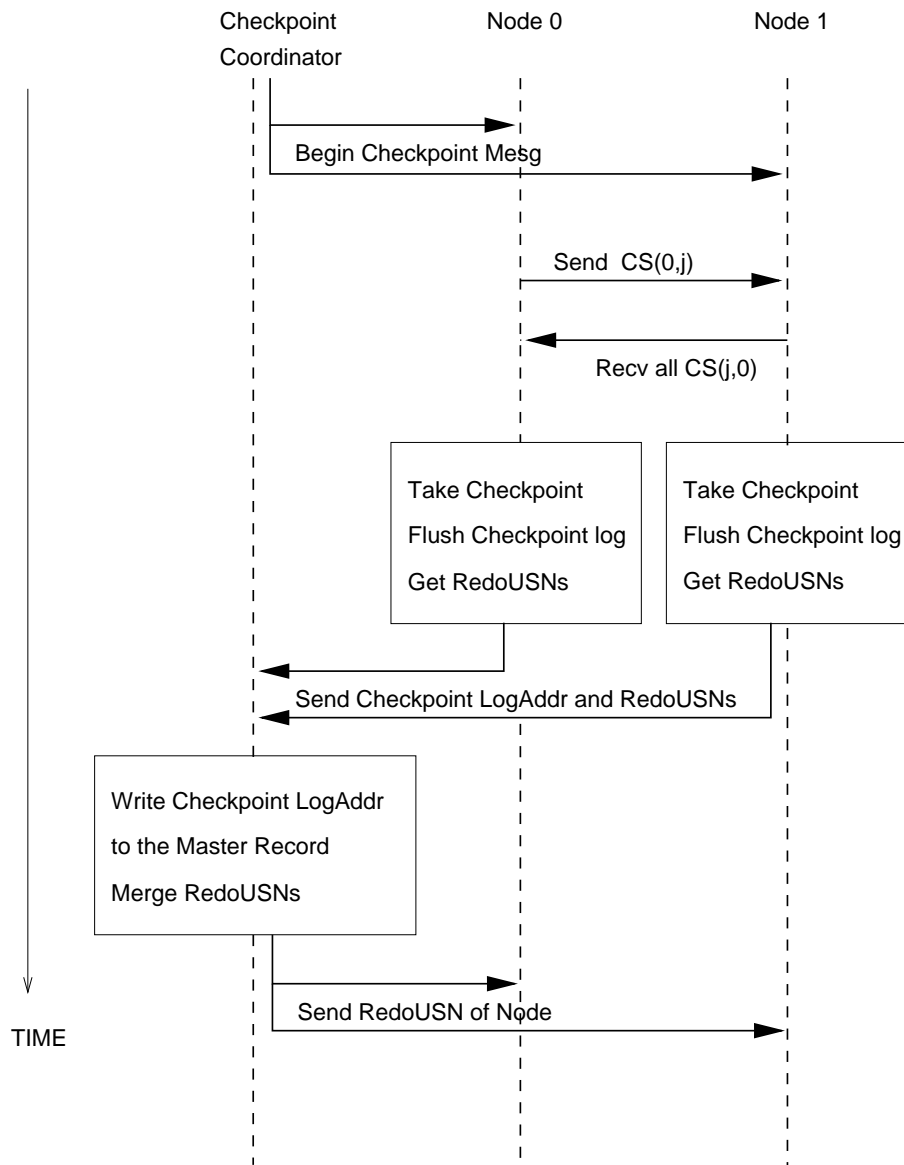


Figure 8.3: Checkpointing Protocol

individual nodes execute independently. Using coordination, we have ensured that pages do not skip a checkpoint by moving between nodes.

Chapter 9

Recovery from System-Failure

In previous chapters we have discussed the implementation of logging, transaction management and checkpointing. In this chapter we will propose the recovery protocol for the system to restart from failure.

9.1 Recovery Coordinator

As seen in the algorithm of *Section 4.3* a lot of synchronization is required. To coordinate the restart recovery activity at the nodes, a Recovery Coordinator (RCVCOORD) thread is started on Node 0. The interaction between the RCVCOORD and OM thread on the individual nodes is shown in *Figure 9.1*.

Function RestartCoordinator ()

Uses:

Returns:

begin

 Get the checkpoint addresses and send to respective nodes

 Wait for all End-Of-Analysis messages and collect the DPTs

 {DistributionTable, newDPTs} \leftarrow Collate entries in the DPTs and redistribute the entries

 Begin log scan of all the nodes

while log exists in the scans **do**

 Get the next log with minimum USN

if log is redoable **then**

 Send log to the node found from DistributionTable

end if

end while

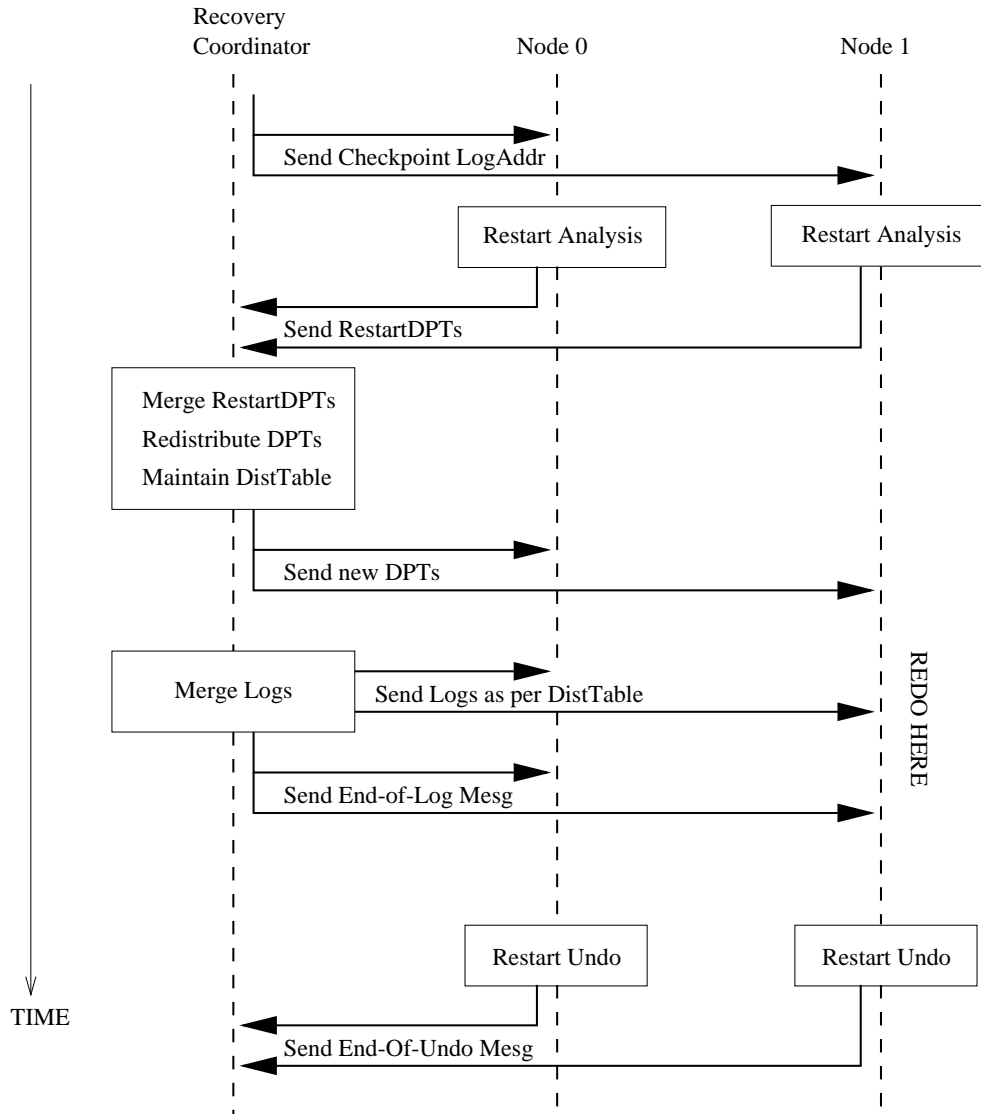


Figure 9.1: Restart Recovery Protocol

Send End-Of-Logs message to all nodes

end

After failure, when *Brahmā* is restarted, it is assumed that all nodes are alive and the OM thread has been loaded before recovery starts. RCVCOORD reads the checkpoint addresses of all the nodes from a well known location on disk, and then sends the checkpoint log address to the OM thread on the respective nodes.

On receiving the checkpoint log address, the OM thread on the nodes perform analysis. Analysis, redo and undo performed by the OM thread during recovery is described in *Chapter 10*. The RCVCOORD waits until analysis is completed on all nodes. It then collects the DPTs returned by the nodes, merges the entries and redistributes the DPTs to load balance the redo work. RCVCOORD sends the new DPTs to the nodes, each of which will contain a set of pages. The node receiving the DPT will be responsible for redo processing on the set of pages.

The RCVCOORD also performs the log merge. It starts reading logs from the redo points on the nodes and processes the log with the minimum USN. If the log is redoable, RCVCOORD probes the distribution table to find the node that will perform redo on the page, and sends the log to it. Redo at the node checks the page before applying the log. When all the logs have been processed, RCVCOORD sends a End-Of-Log (EOL) message to the OM thread on each node. On receiving this message the OM thread starts undo for all loser transactions on the node.

9.1.1 Distributing Redo Work

The purpose of redistribution of pages to the nodes is to load balance the redo work. Load balancing can be seen from two perspectives, viz. I/Os and amount of redo logs applied.

If the pages are equally partitioned among nodes, then I/O load balancing can be achieved. However, if logs received for redo from the RCVCOORD are randomly applied to pages, this is not possible. Instead, when the log is received it is added to a queue maintained for the page to which the redo is to be applied. At the start of redo at a node, only as many queues are scheduled as the number of frames in the Cache Manager. As queues get completed, more queues get scheduled, leading to optimal I/Os.

If the number of processors is large in the system, few or no pages are as-

signed to Node 0, as it will be performing the log merge at the cost of disk I/Os. Log merge will perform sequential log scans and parallelism can be achieved by prefetching log extents as the merge of fetched extents is in progress.

Optimal I/Os need not mean optimal processor utilization. The page distribution could be such that, the number of logs sent to a processor or the number of logs applied to the pages assigned to a node, are few. A partial estimate of the number of logs to be applied per page can be obtained by analysis. Based on this estimate, pages can be distributed among nodes such that the messages sent are nearly equal to all the nodes.

9.2 Summary

In this chapter, we have described the protocol for restart recovery that has been implemented. Restart analysis, redo and undo are discussed in the next chapter. Recovery from single-node failure will be presented later.

Chapter 10

Recovery Manager

The Recovery Manager (RM) which runs on each node uses logs to facilitate crash recovery. In this chapter we will see the components of RM and the routines which perform transaction rollback and restart recovery.

10.1 Initialization

When *Brahmā* is loaded on the processor, restart recovery is first performed to bring the database to a consistent state. RM uses the Object Manager of *Brahmā* to undo any incomplete transactions. Once restart recovery is completed, during normal processing, RM is used to perform transaction commits and aborts.

10.2 Transaction Rollback

As we have seen in the algorithms of OM, updates are performed by latching the page, performing physical update operations, and finally releasing the latch on the page. Once the latch on the page is relinquished, updates are visible to other transactions, which can then perform updates on the same page. This means that it may not be possible to undo the previous update at the same location. For this reason, logical undo operations are written in the physiological redo-undo and NTA logs.

During transaction rollback, redo-undo and NTA logs are logically undone and the undo is logged by compensatory log records. Each log is read, the logical undo operation code is extracted from the log and the appropriate undo routine is called. The undo routines are normal operations of the modules of *Brahmā* which also perform logging. The routines are passed a PFlag with

“Undo” to indicate undo processing.

If new opcodes are added, then the below routine has to be modified to route logical undos to the appropriate undo functions.

```
Error_t RM_PerformUndoOp(Transaction_t* pTrans,
                        LogIterState_t* state, LogAddr_t& UndoNext);
```

Logs are undone upto a user-specified savepoint. If all logs of a transaction are undone, an *end-transaction* log is written, thus completing transaction rollback.

10.3 Restart Recovery

Restart recovery is similar to that in ARIES. However, it is done in coordination with the RCVCOORD as shown in *Chapter 9*.

10.3.1 Analysis

Analysis builds the TransTable as of the time of failure and RestartDPT containing potentially dirty pages. But, analysis on the node is started only when the begin and end checkpoint log addresses are obtained from RCVCOORD. The *end-checkpoint* log record is read to initialize the TransTable and RestartDPT. The OidCounter is also initialized in the HM. The log scan is started from the *begin-checkpoint* log record, or from the start of log (if no checkpoint log exists).

Each log is analyzed as in ARIES. The new LOGTYPE_EXTENTALLOC is treated as a physical update and LOGTYPE_OIDSET is used to set the OidCounter in HM. The LOGTYPE_PAGEFETCH adds the page to the RestartDPT.

Finally, at the end of analysis RestartDPT is sent to the RCVCOORD. As discussed before the RCVCOORD collects the RestartDPT entries and determines the starting point of redo on each node. The RCVCOORD then distributes redo work among the nodes sending them the list of pages followed by the merged logs.

10.3.2 Redo

After analysis, RM waits on the list of pages from RCVCOORD. On receiving the list of pages, the existing RestartDPT is deleted and a new RestartDPT is made from the list.

RM receives redoable logs from the RCVCOORD and the updates are applied. The redoable log types are LOGTYPE_PL_CLR, LOGTYPE_PL_NTA, LOG-

TYPE_REDO, LOGTYPE_PL_REDOUNDO and LOGTYPE_EXTENTALLOC. No new logs are written in this phase.

When a redoable log is encountered and the page is in RestartDPT, redo is to be tried only if the log USN field is greater than the RecUSN stored for the page. The USN check needs to be made with the entry for the node on which the log was written. So RCVCOORD also sends the node identifier with the log. If the above condition is satisfied, the page is fetched from disk and the page USN field is checked.

If the page USN field is greater than the log USN, redo can be ignored. In fact, any log between the log USN and page USN need not be redone. This is indicated by setting the RestartDPT USN entry to the page USN field incremented by one. However, for this USN value (if it exists) we don't know the log address. So a flag is set for the RestartDPT entry to indicate that the log address needs to be filled in when a log of USN greater than RecUSN is encountered.

Function RedoLog (Redoable Log L , ProcessorId I)

Uses: RestartDPT

Returns:

begin

Get the RestartDPT entry E for page P of redoable log L

if E is valid and log USN > $E[I].\text{RecUSN}$ **then**

if $E[I].\text{flag}$ is set **then**

 update $E[I].\text{RecUSN}$ and log address for $E[I]$

end if

WriteLatch the page P

if Page USN field < log USN **then**

 Redo the update

 Set page USN field

else

$E[I].\text{RecUSN} \leftarrow \text{page USN} + 1$

 Set $E[I].\text{flag}$

end if

UnLatch P

end if

return

end

The redo action stored in the log may be a physical or logical operation. The logical redo routines are normal operations of the modules of *Brahmā* which are prevented from logging. The routines are passed a PFlag with “Redo” to indicate redo processing.

If new opcodes are added, the below routine has to be modified to route logical redos to the appropriate redo functions.

```
Error_t RM_PerformRedoOp(OneByte_t* buff,
                        LogIterState_t* state,
                        Disk_t diskNo, Page_t pageNo);
```

10.3.3 Undo

Restart redo repeats history. After redo, the pages of the database can be considered physically consistent. History may contain updates of incomplete transactions that will be reflected on the pages. So, undo consults the TransTable to rollback loser transactions. At each node in a single sweep of the log in the reverse chronological order, undo is performed.

The maximum USN of the next undoable log addresses for the transactions in TransTable is chosen to be undone. When all the operations of a transaction are undone the *end-transaction* log is written. This is continued until the TransTable contains no loser transactions.

10.4 Summary

In this chapter we have seen the module added to *Brahmā* to perform transaction rollback and restart recovery. The algorithm of recovery is ARIES, with simple modifications to fit *Brahmā*'s requirements.

Chapter 11

Recovery from Single-Node Failure

In *Chapter 9* we have seen the restart recovery algorithm from system-failure. In this chapter we will discuss the issues in recovery when a single node fails. The easiest option is to fail all nodes and restart the whole system. But our aim is to have data, as well as maximum nodes, in the system available at all times.

11.1 Assumptions

Failures in the system would be detected when messages fail to reach the destination. Messages could fail due to two reasons – either the interconnection bus fails or the node is not responding. Bus failure is a rare event and will anyway cause a system-failure. So we concentrate only on the failure of nodes.

We also assume that during recovery of one node, there is no other node failure. After the already failed node has been recovered it is feasible to recover from another node crash.

11.2 Detection of Failure

When a Node i fails in sending a message to Node j , it can be assumed that Node j is dead. This is also confirmed by Node i with the other nodes in the system. The alive nodes coordinate and elect a recoverer for the failed Node j . Let this node be Node r . While recovery is in progress all requests to the failed node are frozen and not processed.

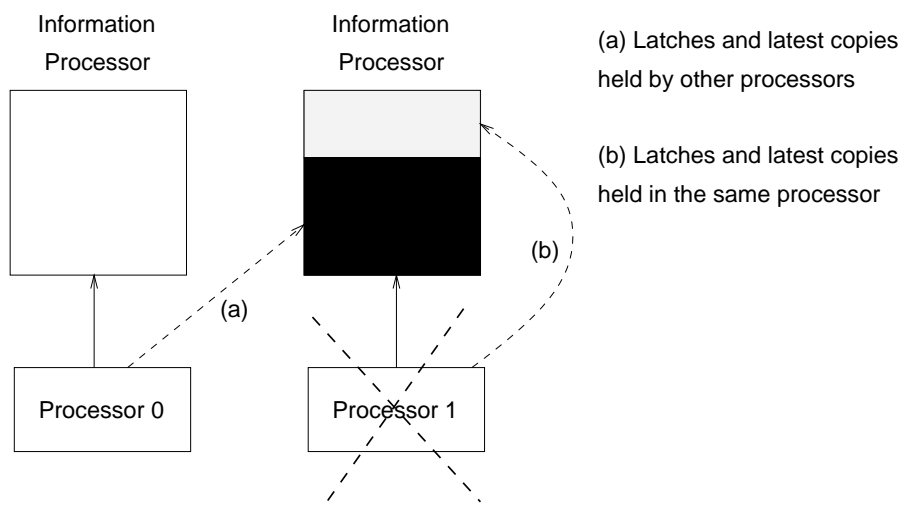


Figure 11.1: Lost Information of the IP on crash

The simplest thing that can be done is sending a reset signal to the Node j through the OS. This signal will reload the software on the node and at startup the node can perform restart recovery. Though this seems simple and attractive, it has two problems. The node may not respond to the reset signal due to hardware failure or the node may take too much time to restart during which transactions on the other nodes may have to wait.

Instead we discuss an algorithm in which the Node r will perform restart recovery on behalf of the failed node and also reset it. This algorithm can also be performed by the failed node if it is reset and comes alive in due course.

11.3 Information Processor

The failed node had an IP which maintained ownership information for a set of pages that were assigned to it. This information is lost when the node fails. The first task is to rebuild the IP of the failed node, by sending a message to the alive processors to give information of the latches they hold. Once the information is obtained, the IP can be partially rebuilt. It can be completely rebuilt only when latches held by the failed node and latest copies in the various caches are detected. This is shown in *Figure 11.1*, where the dark part represents latches or latest copies held by other nodes, and the shaded part represents latches or latest copies of pages held by the failed node.

To obtain information of the latches held by the failed node restart analysis needs to be done. At the end of analysis, the superset of the pages on which

latches may have been held by the failed node, can be obtained.

Also the IP on the failed node has to be restarted. The easiest thing would be to start a proxy IP on another node, which will accept all requests of the failed IP. But this may cause some load imbalance in the system. Therefore each IP is reconfigured to share the load of the failed IP.

11.4 Algorithm

With the recoverer (Node r) elected, the recovery of failed Node j can be done as shown below. The detailed explanation of the steps is done in *Section 11.4.1*.

Function RecoverNode (Failed Node j , Recoverer Node r)

Uses:

Returns:

begin

- (1) Send message to all IPs to reconfigure them. Any request coming to the IP of failed Node j will be redirected to another IP; but, these requests are blocked until restart analysis is done.
- (2) Since recovery of failed Node j is done by Node r , all CM requests to be sent to Node j are instead sent to Node r . Until the end of the redo pass, no requests can be made to the CM of Node r on behalf of Node j .
- (3) Node r sends a request to all CMs for latch entries that map to the failed IP. This can partially rebuild the IP as described in *Section 11.3*. The entries are inserted into the existing IPs as per the new configuration.
- (4) Spawn the LGM for Node j on Node r and perform restart analysis.
- (5) After restart analysis, RestartDPT contains the set of pages that could potentially be inconsistent. The IPs at the other nodes are probed to eliminate pages from RestartDPT whose latches or latest copies are held by some Node i , where $i \neq j$.
- (6) IP at the failed node must also contain entries of the pages whose latest copies are in the cache of the alive nodes. A latest copy can be determined only by comparing the page with the disk copy and the last update USN of

the page entry (if it exists) in RestartDPT. The page can be removed from RestartDPT if some node has a latest copy.

- (7) Find pages owned by Node j from IPs and add them to RestartDPT.
- (8) Once RestartDPT is constructed, as shown in Steps (5) , (6) and (7), IPs are updated with entries from RestartDPT to make Node j the latch holder of the pages. Now requests to the failed node's IP entries can be enabled.
- (9) Perform restart redo for the pages in RestartDPT. Checkpoint logs are obtained for all nodes and consulted to find redo points. Logs are merged and redos are applied.
- (10) Update the IPs to change ownership of pages held on behalf of Node j at Node r , and allow latch requests to the pages held by the failed node.
- (11) Perform undo of the loser transactions.
- (12) Reset the Node j , reconfigure the IPs and take a checkpoint.

end

11.4.1 Discussion

As soon as node failure is detected in Step (1), all the nodes are informed by the recoverer to reconfigure their IPs in order to distribute the load of the failed IP. Pages that mapped to the failed IP may be incorrectly shown latest on disk if requests are not blocked until the failed IP is rebuilt. So all requests to the failed node's IP are blocked until the end of analysis pass.

Similarly, as shown in Step (2), the CM of Node r will act as the CM of failed Node j during recovery. Any node getting the IP entry and finding the page with the failed node can get it from Node r , though Node r will return the page only after redo (when the page is consistent). Another approach to solve the problem is to block the page requests at the IP. The IP can check ownership of the page on a latch request and on finding it to be owned by the failed Node j , block until it gets the end-of-redo message. After this it will update ownership and allow page requests to Node r .

In Step (3), once the CMs of the alive nodes enumerate the latches held,

the IP is partially rebuilt. It is not complete until the latches held by the failed node and the latest copies held by nodes in their respective caches (without latches) are recorded. But, to detect this analysis needs to be done at the failed node as shown in Step (4).

After analysis, we have a list of pages in RestartDPT which could be potentially in the latched state at the failed node during failure. If some Node i holds the latest copy of a page listed in RestartDPT, the page entry can be discarded from RestartDPT in Step (5). That is, the page need not be brought upto date in redo.

We need to find out the pages whose latest copies exist (not latched) in the cache of nodes and map to the failed IP. These pages are obtained from CM of the alive nodes in Step (6), along with their USNs. If more than one node holds the page, the node with the maximum USN in cache, say Node i , can be deemed owner of the latest copy. But, this may not be the case as the page may have moved to another Node m , where it was updated, and then flushed to disk. Under such a scenario the page at Node i is a stale copy. To check this, the page is read from disk and its USN is compared with the maximum USN to decide the owner.

The page entry in RestartDPT need not mean that the page was latched at failed Node j . It could be that the page was updated at Node j , after which the page moved to a Node i where it was updated, and CM at Node i holds the latest copy of the page. If the last update USN of the page at failed Node j is lesser than the USN of the page at Node i , the RestartDPT entry can be discarded. However, latest copy ownership can only be decided as per the disk copy of the page.

RestartDPT is built on the basis of logs. If a page P moved from Node i to Node j (which failed), and no log was written at Node j , page P will not be added to the RestartDPT. Such pages can be detected by asking the IPs on the nodes to return entries whose owner is Node j . The pages which do not exist in RestartDPT are added in Step (7). But since the IP of Node j itself has failed, some information is lost. Solution to this problem is discussed in *Section 11.4.2*.

After all the above steps, RestartDPT consists of the list of pages which need to be brought upto date in redo as per the logs. The ownership of all pages in RestartDPT is set to the failed Node j .

The contents of the RestartDPT and IP at Node j before redo are:

$$\text{RestartDPT} = P \cup \{R - Q\}$$

where, P is the set of pages whose latch or latest copy is held by Node j as per IP_i ($i \neq j$), R is the set of pages found in analysis, and Q is the subset of R whose latch or latest copy is held by CM_i ($i \neq j$). And,

$$IP_j = M \cup N$$

where, M is the set of pages (mapping to IP_j) whose latch or latest copy is held by CM_i ($i \neq j$), and N is the set of pages in RestartDPT mapping to IP_j .

The pages in RestartDPT could have unflushed updates of actions performed at various nodes. The checkpoint logs of all the nodes are obtained and the logged DPTs are read. From the logged DPTs, the entries of pages in RestartDPT are extracted, and merged to get the minimum redo point. The minimum redo point cannot be greater than the *begin-checkpoint* log at a node. If no checkpoint exists then the minimum redo point is the start of logs at all the nodes. Logs are merged from the redo points at the nodes and history is repeated. Latch requests are not sent to the IP during redo; instead the page is directly read from disk, if not found in the cache of Node r .

After redo, the ownership of pages in RestartDPT are updated in the IPs. So page requests can now be sent to Node r if pages are still cached there. Loser transactions are undone finally.

As recovery is in progress, the failed Node j can be reset through the OS. This will reload the software if the node has no hardware failure. After the node starts up, it is made to wait for a message that will let it continue the recovery in progress. At the end of each step Node r attempts to send a message to Node j , failing which it continues the next step.

Finally when recovery is done, a checkpoint is taken. The checkpoint for Node r , may contain some of the pages that were made consistent by redo. Node j will contain an empty checkpoint; that is, it will not have any pages in the DPT and the TransTable will be empty.

11.4.2 LOGTYPE_PAGEFETCH Log

A page is added to RestartDPT only if the logs contain an update to the page. Sometimes, a page may move in with dirty updates from another node, but no updates are performed at the failed Node j . So in Step (7), the IPs are checked to find pages owned by the failed Node j and they are added to RestartDPT if necessary.

But, there still exists the problem of the incomplete IP of the failed node. This IP cannot return information about the latest copies of pages held by the

failed node itself. To solve this problem, a LOGTYPE_PAGEFETCH is written whenever a page is requested by a Node i from some other node, such that the IP of the page is also on Node i . If writing the log fails, it is ensured that the page is written to disk before it is shipped.

11.5 Summary

In this chapter we have discussed the design of single-node failure recovery. Though many of the modules have not been implemented, the basic infrastructure is available for the remaining implementation.

Chapter 12

Implementation and Tests

The modules of *Brahmā* are implemented in the C++ programming language. Since, the *Anupam* machine is not available, the OS Layer is being simulated on *Solaris 2.4*. The message passing primitives are implemented by socket calls, and the disks are read/written from UNIX files.

The implementation is made as flexible as possible by means of a configuration files. The configuration file is used specify, among other things – the path of the UNIX files that are used as disks, number of processors and disks, disk size as number of pages, page size as number of bytes, and cache size as number of pages. In *Anupam*, these parameters will be available from the OS Layer as constants.

Each processor is run as a process on *Solaris*. For each process the configuration file consists of the processor number. This is the only parameter that is unique per process in the configuration file. As the processes communicate among each other via sockets, the configuration file also includes a list of host-names and socket-ids that can be used to send messages to other nodes.

On each node we ran OM-tests to check the creation, deletion and updation of objects on disks. There were two kind of OM-tests, one for objects to be created within clusters and the other for objects without clusters. Each OM-test did the following: (1) Create some 79-byte objects, (2) Delete alternate objects (3) Create some 54-byte objects, (4) Reduce every fourth 54-byte object to 29 bytes, (5) Finally, increase every third 79-byte object to 104 bytes. At the end of each step, the objects in the system were retrieved and their contents checked. Each OM-test was run as a separate transaction and 20 such tests (10 of each type) were run simultaneously at each node.

The OM-tests were carried out with varying configuration parameters like – number of disks, number of processors, and cache size. Transaction rollback

was tested by aborting some of the tests and checking the state of the database. Restart recovery was tested by failing the processes. Checkpoints were occasionally taken and the contents of the checkpoints seen. At the end of each test the state of the database was checked by a utility.

Currently *Brahmā* is being used in other applications, like *Garbage Collection in OODBs*, where stress testing has been done with large number of objects and huge logs.

Chapter 13

Conclusion and Future Work

In this dissertation we have reviewed the implementation of recovery in *Brahmā*. We have built a log manager which serves in writing and fetching logs. Restart recovery has been implemented for multiple nodes, with fuzzy checkpointing.

13.1 Comparisons with Related Work

[MN91] is one of the very few papers published in the field of recovery in SD environments. The primary motivation of this paper is to solve cache coherency problem for SD, integrated with concurrency control and recovery. Many of the subtleties of restart recovery are also discussed in detail. Some comparisons with our scheme are as follows:

- [MN91] performs logging per node as we do in our system. They have a log merge process in one of the nodes which has access to all the logs in the system. The same can be implemented in our system; instead this process is spawned in our system when required. But one important difference is that in [MN91], nodes are assumed to have synchronized clocks and so timestamps appended to the logs can be used to order them. In our system there is no such assumption and we use USNs for this purpose.
- The page latch information in [MN91] is handled in a two-level fashion as explained in *Section 2.4.4*. In our design, with the IP architecture the latch information is distributed on all nodes. Each IP maintains the owners of the latch or latest copy of the page. Each transaction asks the CM for a latch which can be got with the ownership information obtained from the IP.

- Since the logs are distributed and checkpoints obtained are separate, [MN91] also faces the problem of pages escaping out of the CM during the checkpoint. So they checkpoint the GLM table which maintains the min RedoLSNs of the nodes. In our system, we perform synchronized starting of checkpoints, but pages moving between nodes are also noted in the checkpoint. That is the checkpoint does not freeze the movement of pages when it is in-progress.
- Single node failure in [MN91] seems to assume that the GLM never fails. The GLM holds vital recovery information regarding pages and its failure along with the failing node can cause problems. This is not addressed in the paper. Although in our design IPs are distributed, we have presented a comprehensive algorithm for rebuilding the IP on recovery from a single node failure.
- [MN91] hints that analysis and redo recovery can be performed simultaneously. The global checkpoint log is read to find out the pages and their RecLSNs. The minimum of RecLSNs in the checkpoint DPT and the *begin global checkpoint* record can be taken as the redo point. The same can be achieved by our scheme. Redo merging is begun from the RedoLSNs of pages in the checkpoint of the nodes. As merging of the logs is done, (1) the logs are analyzed, and (2) the updates are redone by sending the logs to the nodes designated for redo.

13.2 Overview

Although recovery has been implemented in *Brahmā* on the lines of ARIES, some of the novel features of our scheme are:

1. We support the *no-force* and *steal* buffer management policies. The granularity of locking can also be made as fine as objects in the pages.
2. *Separate logs* are maintained for the nodes. The logs are merged during restart redo for system and single-node failure.
3. Each page maintains the *update sequence number* instead of a LSN. The USN behaves as a timestamp, and is also stored with the log. Logs are ordered across nodes using the USN. So the space overhead is not increased.
4. The changes made to pages are logged *physiologically*. This saves a lot of log space.

5. *Minimum data overhead* while shipping the page is required, as only the DPT entry is shipped along with the page. This entry will be a few tens of bytes (depending on the number of nodes) as compared to the page size of kilobytes.
6. Transaction management in *Brahmā* supports *partial and total rollback*. Users can establish savepoints and perform partial rollbacks to the savepoints.
7. Redo work is *equally distributed* among nodes during restart recovery for load balancing. This is done by partitioning the pages that require redo among the nodes.
8. *Fuzzy checkpointing* is started synchronously at all the nodes, and checkpoint logs are combined to find the restart redo point in case of failure.
9. *Single-node failure recovery* is achieved without blocking the other nodes in the system. Normal processing, that does not involve the failed node can be continued on the other nodes, even while recovery is in-progress.
10. During single-node failure recovery the failed node is *reset* and it tries to take over the recovery process. If this fails, the recoverer node continues recovery and system resumes normal processing without the failed node.

13.3 Future Work

The basic recovery modules implemented in *Brahmā* have transformed a typeless object storage manager into a powerful system for multiple users with data durability and atomicity. This system can act as a stepping-stone for solving many research issues in parallel databases. The parts of transaction management that remain to be studied and implemented in the system are as follows:

13.3.1 Transaction Manager

- The multi-processor architecture of the system must be exploited to provide data as well as program parallelism. A transaction must be given the facility to *spawn* sub-transactions at other nodes for function shipping.
- For transactions running on many nodes the *2PC protocol* is to be provided for distributed commits.

13.3.2 Single-Node Failure Recovery

- We wish to extend the single-node failure recovery algorithm to handle more than one failure at a time.

13.3.3 Lock Manager

To complete the properties of transaction management we need to provide concurrency control. A typical implementation of concurrency control is a lock manager. A primitive lock manager is available for a single-node system. This needs to be extended as:

- The lock manager will be *distributed* on the nodes of the SD environment. This will help remove the bottleneck in the system due to a single lock manager. Also, when a lock manager at a node fails, the other nodes must be able to resolve the locks held by the node before crash. This is to make sure that the resources held by the crashed lock manager are recovered consistently.
- The lock manager will help define a *locking hierarchy*. The locking hierarchy will help getting intention locks at higher levels, and thus reduce lock overheads.
- *Adaptive locking* will help in reducing locks at a fine granularity by holding locks at a coarse granularity unless there is a conflict. The hierarchy of adaptive locking is a root containing the entire database, and as the tree grows the granularity of locking is made finer.
- The lock manager must resolve distributed *deadlocks*.
- The lock manager must be able to provide locking facilities for different degrees of isolation. Information from the log manager and transaction manager can be used to check if locking is necessary at the required degree of isolation.

References

- [Des96] Ajay Deshpande. Cache Coherence and Implementation Issues in a Typeless Object Manager. Master's thesis, Dept. of Computer Science and Engg., Indian Institute of Technology, Bombay, 1996.
- [DG92] David DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.
- [EN94] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Benjamin Cummings, 1994.
- [Gog96] Sadanand M. Gogate. Disk Management in a Parallel Database System. Master's thesis, Dept. of Computer Science and Engg., Indian Institute of Technology, Bombay, 1996.
- [GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [Khi96] Amit Khivesara. Operating System and Storage Issues in a Parallel Database System. Master's thesis, Dept. of Computer Science and Engg., Indian Institute of Technology, Bombay, 1996.
- [KS91] Henry Korth and Abraham Silberschatz. *Database System Concepts*. McGraw-Hill International, 1991.
- [Lom92] David Lomet. MLR: A Recovery Method for Multi-Level Systems. In *Proc. ACM-SIGMOD Conference*, pages 185–194, 1992.
- [MDRK94] S. Mahajan, P. Dhekne, K. Ramesh, and H. Kaura. ANUPAM - A High Performance Parallel Computer. *Computer Science and Informatics*, 24(2):33–43, 1994.

- [MHL⁺92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.
- [MN91] 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. Technical Report RJ 8017, IBM Almaden Research Center, 1991.
- [MN94] C. Mohan and Inderpal Narang. ARIES/CSA: A Method for Database Recovery in Client-Server Architectures. In *Proc. ACM-SIGMOD Conference*, pages 55–66, 1994.
- [MNP90] C. Mohan, Inderpal Narang, and John Palmer. A Case Study of Problems in Migrating to Distributed Computing: Database Recovery Using Multiple Logs in the Shared Disks Environment. Technical Report RJ 7343, IBM Almaden Research Center, 1990.
- [Moh90] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. Technical Report RJ 7344, IBM Almaden Research Center, 1990.
- [MPTW94] C. Mohan, H. Pirahesh, W. Tang, and Y. Wang. Parallelism in relational database management systems. *IBM Systems Journal*, 33(2):349–371, 1994.

Acknowledgments

My guide, **Dr. S. Seshadri**, has been inspirational throughout my education at IIT. His invaluable guidance, through suggestions and comments, was the primary motivation in the project. The second driving force behind the *Brahmā* project has been **Prasan Roy**. His depth of theoretical knowledge coupled with hard work proved awesome at times. He has always been there to discuss problems I was facing and help search for solutions.

A special thanks to **Dr. S. Sudarshan** for his encouragement and for helping me with references. **SIGDB** was instrumental in helping me understand ARIES and finally grill my implementations of multi-node recovery. So, a special thanks to all who sat patiently listening to me. Many of my peers have helped me through the highs and lows of life at IIT. A special mention of **Aalop, Amish, Gopal, Hardeep, Manish, Prasan, Samir, and Srikanth**, for making it some of the best years of my life.

Finally, I would like to dedicate this work to my parents, **Shyamala Parameswaran** and **P.S. Parameswaran**, who did everything to make me this worthy.