An Implementation of Transaction Logging and Recovery in a Main Memory Resident Database System

Abstract

This report describes an implementation of Transaction Logging and Recovery using Unix Copy-On-Write on spawned processes. The purpose of the work is to extend WS-Iris, a research project on Object Oriented Main Memory Databases, with functionality for failure recovery.

The presented work is a Master Thesis for a student of Master of Science in Computer Science and Technology. The work has been commissioned by Tore Risch, Professor of Engineering Databases at Computer Aided Engineering laboratory (CAElab), Linköping University (LiU/LiTH), Sweden.

Keywords

Transaction logging, logical logging, recovery, main memory database, copy-on-write, process forking

Jonas S Karlsson
CAElab, IDA,
Linköping University

CHAPTER 1

Introduction

In this section, an introduction to theWS-Iris database system and motivation for the work is given.

1.1 WS-Iris

WS-Iris [Litwin-92] is a memory resident object oriented database system written by Tore Risch at the Database Technology Department, Hewlett-Packard Laboratories.

1.2 Reason & Goal

The aim of this project is to enhance WS-Iris with recovery functions, which will be responsible for recovery after a crash.

The reason for building a recovery-system is to secure data storage in the database. WS-Iris is a main memory resident database and thus there is no database on the disk that is constantly updated. Benefits thereby gained are tremendous at search and update time. The overhead for handling data can be minimized compared to a disk-based system. However, main memory is not stable memory, and the operating system is not stable. Crashes will occur, and information stored in the computers memory will be lost forever. Data has to be saved (backuped) elsewhere, and this is the primary concern of this work.

The main causes for crashes are programming errors, operating system faults and hardware failures. These failures often lead to uncontrolled process termination. This work is focused on handling data that has been committed to the database, and making that data persistent.

1.3 Contents

Chapter 2 introduces the WS-Iris programming system. Aspects of WS-Iris directly concerned with the problem are described and exemplified.

Chapter 3 provides a short survey of well known methods used in research and commercial systems for recovery processing.

In chapter 4 a specification is presented concerning the requirements of the work and the purpose of the work.

Chapter 5 presents the chosen implementation and justifies the design decisions.

Chapter 6 evaluates the implementation and the achieved results. Possible improvements are also briefly discussed.

A compiled list of technical terms used in this thesis may be found in appendix A.

CHAPTER 2

WS-Iris

This chapter describes functionality already present within the WS-Iris system that are of use when implementing a Recovery System. Further information about the Iris DBMS can be found in *Object-Oriented Concepts, Databases, and Applications* [Fishman-87].

2.1 Internal Workings

WS-IRIS is written and optimized with memory residency in mind. It is implemented in C giving a special version of Common Lisp under which Database Queries are executed using a transformation into ObjectLog that is interpreted by the Lisp. At the prototype stage, WS-IRIS was entirely written in Common Lisp but has later been migrated towards an implementation in C for reasons of efficiency, the interactive parts are still written with Lisp as a basis, however.

Since WS-IRIS partly is implemented in C, and partly implemented in the Lisp that it implements, there is a powerful ”foreign” function interface utilities. This makes it possible to implement OSQL (Object Symbolic Query Language) and Lisp functions in C code.

OSQL is used as the main database query language, and it uses the concepts of types, objects, and functions.

2.1.1 The self-contained Lisp

Lisp is good for prototyping and development of a system. One reason for not using C in this work, is that data is stored and created in the Lisp and therefore must be accessed using lisp primitives. This would mean that the C-code would not be plain C.

So, since most structures and operators are easily accessible from Lisp, but only with effort from within C (one would effectively be writing Lisp in C) there are clearly advantages of using Lisp for implementation of algorithms.

2.1.2 Foreign functions

Foreign functions are used as a means of extending the Lisp/OSQL language with various primitives, usually for interfacing to the operating system and outside world. This is further explained in *WS-Iris; A Main Memory Object Oriented DBMS* [Risch-92].

2.2 Image

The data area called *image* in the WS-IRiS system contains all of the database and interfacing information for the database implementation. When the image is stored on disk it is called a dump. When WS-Iris is started, a dump (amos.dmp) is automatically loaded.

In WS-Iris there are functions for saving and loading a dump of the database. These functions are available from OSQL, Lisp as well as from C. An important issue is the consistency of the dump: a dump should only be performed on a committed database. This is ensured when the routines are called from OSQL, but care has to be exercised when they are called from Lisp. The dump also contains references to foreign functions. These are relocated at load-time. Lisp functions and variables available at save-time are also contained in the dump.

2.3 Logging (Histories)

For logging purposes, there are history functionality that can be used. The data stored by the history function are undo and redo information specific for current, non-committed transactions. This information can be used at commit-time for creating a log.

CHAPTER 3

Survey on Recovery

There has been quite a lot of work done in the database recovery area, mainly because of the fact that most database system still are managing databases that resides on disk. Lately, however, there has been an amount of research on MMRDB (Main Memory Resident Data-Bases), and an identified problem is that of securing persistency. Different systems for research on a variety of recovery algorithms has been written, *Principles of Transaction-Oriented Database Recovery* [Haerder-83] contains interesting material on the area of Recovery.

3.1 Introduction

A MMRDB takes advantage of the residency fact, and can therefore achieve astonishingly fast results for creating and searching data. Also updates can be made at the same high-speed. But there is a problem, since the data is stored in main memory it can also be lost quite easily. Therefore, in order to make sure that the changes made are persistent, they are often saved onto disk. *Persistency* means that the changes made to the database somehow can be recreated. It is often a a requirement that *committed* updates are persistent. The simplest way of accomplishing this is by using a log-file on disk on which, at each commit, the updates are written sequentially at the end.

Having a log-file is sufficient, but not very economical. The time used for recreating the database, bringing it up-to-date using the log-file, will be linear to the size of the file. Therefore different strategies has been developed, called *Checkpointing strategies*. The obvious optimi-

zation is to try to keep a copy of the database image updated on disk. Then the time used for recovery would be limited by the size of the saved image. But then, keeping the image on disk updated would be extremely time-wasting. In fact, the only thing achieved would be a Main Memory Cashed DataBase, and much time would be spent on trying to cluster connected data.

Many checkpointing schemes therefore make use of both an image-file and a log-file, balancing somewhere inbetween the two extremes.

Checkpointing can be divided into two categories, *monoplex* and *duplex*. Monoplex strategies maintain only one backup database. Duplex strategies maintain two. Duplex strategies generally requires more I/O for backup, but has the advantage of being safer with respect to media failures since there are two backups. When using a monoplex strategy consistent checkpoints can not be made, since, if a current checkpoint can not complete, this may lead to an inconsistent backup. Only *fuzzy* checkpoints can be used with monoplex backups [Salem-90].

Upon designing a recovery system for a MMDBS one can note that the only need for disk I/O is for backup/recovery purposes, and it can therefore be tailored solely for the checkpointing. One observation is that disk I/O on larger blocks is more efficient, thus the checkpointer has to wait longer but that does not stop the application transaction processing since the transactions are not dependent on completion of data writes [Garcia-Molina-92].

3.2 Saved Images

As noted earlier, one important requirement of a database is that of securing the existence of a complete and consistent saved image. This is important for persistency reasons. There are quite a lot of different methods. A short survey of some well known methods follows.

3.2.1 Ping-Pong (Duplex strategy)

”Ping-Pong-Ping-Pong-Fault-Serve-Pong-Ping-Pong-Ping”. This can serve as an good description of the duplex strategy system. Ping and Pong relate to the saving of images at two different places. Fault relates to the uncontrolled process termination. Serve relates to the recovery process.

The idea is that images are saved at two alternating places, and between the alternating savings logging is done of all changes made to the database. The saving of a new image could for example be triggered at a certain time interval or by the size of the log. Usually saving is started at well-defined moments such as at commit-time. In case of an crash, when the current image is not successfully saved, the previous alternative image can be used. This always ensures a consistent image, in a simple way using duplication.

3.2.2 Fuzzy Checkpointing (Monoplex strategy)

System M [Salem-90] implements a variant of fuzzy checkpointing that they call FUZZ.

This monoplex strategy starts with a mark in the log that a checkpointing has been started. Thereafter the part of the image that is *dirty*, that is has been changed since last checkpoint is saved during continued transaction processing. After all the dirty parts have been saved on disk, information is recorded into the log that the Checkpointing was successfully completed.

The method is page-based and requires interceptance on each write or update of a page for setting a dirty-bit. For efficiency this should be supplied by hardware.

3.2.3 Black/White Checkpointing

This checkpointing strategy [Salem-90] is based on a schema that saves all pages onto the disk. Pages marked dirty are written onto the disk, processed in an unspecified order. After all pages have been processed the transaction log is saved on disk, and now also holds information about all pages that have been modified after being saved onto the disk. Since there are no ordering constraints upon the page saving, transaction processing can continue during the time the backup is being made. However, there are some requirements that has to be fulfilled, otherwise transactional consistency cannot be maintained. One of these is that no transaction may involve both pages that have been saved and pages that have not. If so the transaction has to be aborted.

Black/White checkpointing also requires the implementation of a dirty-bit. Locking of single pages is required so that the saving of a page is treated atomically.

3.2.4 Copy-on-Update Checkpointing

In copy-on-update checkpointing [Salem-90], a fully consistent snapshot is maintained in memory. The checkpointing starts at a moment where the database is consistent. Thereafter normal transaction processing is allowed to continue. When a transaction updates a page that has not yet been written onto disk it updates a copy of that page while the original page is kept until it has been saved onto the disk.

Two problems can be identified. First, if a transaction's updates span over many pages, the entire database might have to be copied. Second, initially one has to start in a consistent state, thus possibly waiting for all transactions to complete and not allowing any new transactions.

One advantage is that the checkpointing cannot cause transactions to abort.

3.3 Logging

Another requirement is that of durability: once a transaction has been committed it must be guaranteed that the result is protected from system failures. This is usually done by logging transactions and the data that they modify. Different strategies are being used in different systems. The most common strategy is to have some sort of *stable* memory, often referred to as *nonvolatile* memory. This is often of limited size and implemented by battery backuped CMOS RAM.

Logging can be implemented by storing old and new information, also called undo and redo information. Abortion is then only a matter of restoring old values in the reversed order.

There are several different levels that could be chosen for the actual log-information. A change in the database can be described either using information about the *effect* of the change or the *operations* with operands. The effect is the actual change in the bit-pattern in the memory, and the operations are more concerned with the semantics of the update.

3.3.1 Physical Page Logging

Upon changes, the whole of the page with the changes applied is saved in the log. This is a common method in disk-based commercial

systems [Haerder-83]. Most systems that use this method has a page-based memory handling system, with hardware support for detecting writes to pages.

Clustering of data is almost a requirement of this method, since if data is spread on many pages a simple update of the database could, in the worst case, cause the whole database to be written onto the log. Also, very small updates would cause at least a whole page to be written onto disk which introduces more overhead than needed.

3.3.2 Physical Transition Page Logging

A more storage efficient method than physical page logging is transition ditto. It stores only the differences between the old and the new page. This is really an exploitation of the binary exclusive-or operator (apply twice and you are back).

By viewing the difference as a bit string, when small changes are involved, there will be lots of 0’s in it and there are well known methods for compressing such bitstrings.

This method is more an optimization of the Physical Page Logging method, but as a sideeffect the requirement of clustering of data can be relaxed.

3.3.3 Transition Access logging

In transition access logging, logging is done on entries of storage structures. Operations are typically adding, deleting and replacing. This kind of logging is often ruled out because of the requirements of an indirect scheme [Haerder-83], but is a more attractive approach on the next higher level.

3.3.4 Logical Logging on Recordlevel

At the Logical Logging level, updates of the database will be a composition of updates of tuples or records. That means that only the operations (Insert, Update and Delete) and their respective data is written to the log. Undoing operations from the log is then a matter of using the reverse operation for each log entry. This is a very storage efficient method [Haerder-83].

In this case clustering of data is not required.

CHAPTER 4

Requirements

There are several methods and several different safety levels that can be used in design of a recovery system. The main objective is to be able to recover after incorrect database program termination.

This work has the following short specification:

- Ping-pong
- Do image-save in background
- Several ways of triggering image-saving
- Logging
- Readable log-file
- Optional background log-save

4.1 Important Facts

Some requirements and properties regarding WS-Iris limits the possible solutions. They are:

- Object Oriented Memory Manager.
- Main Memory Residency.
- Histories with new and old information of atomic operations.
- Portability.
- No concurrent transactions in WS-Iris.
- HP/Sun-Unix: fork uses copy-on-write.
- Kernel source for WS-Iris not fully available.

- Research system, exploring functionality

The object memory manager does not cluster data and thus lets the queries and updates fully make use of the main memory residency. A history is kept for abort implementation. The history list contains atomic actions that has been executed in the current transaction. Old and new information is stored efficiently using pointers.

The system is currently available for HP and SUN workstations, thus eliminating the use of any specific hardware architecture features such as nonvolatile memory. The only feature that can and should be used is that of sharing memory between two executing processes using the unix command *fork*. Fork on HP-unix (and recently implemented on) SUN-unix uses *copy-on-write*, thus enabling efficient snapshots to be made.

Since the source code of the WS-Iris kernel is not available, the types of changes that can be made to the WS-Iris system, on for instance the Memory Manager level, are limited.

4.2 Main idea

Nearly all approaches in research and implementations in the area of crash recovery for databases are based on the concepts of an image and a log. The image holds a *snapshot* of the database. The log holds information about updates to the database that would have to be applied to the snapshot in order to update it to the current state, in the event of a crash. Variations on the theme of logging are mostly on how the log physically is materialized.

The prime concern of variants on the image-and-log-method is efficiency, either that of disk-activity minimization or that of recovery time minimizations.

4.2.1 Image

The goal of a recovery system is that there should be a consistent image available for restarting, either directly or indirectly, by means of some sort of log.

Incremental updating

A simple approach would be that of updating the saved image via some scheme. A variant of this was implemented by Levy [Levy-92]. One drawback with this, however, is that while this methods minimizes the log, updating has to be done in place. When this is considered, one realizes that it will be heavy on the positioning side, data will almost for certain be scattered all over the image and thus also in the file. Surely there are functions in most operating system that optimizes disk-access order but these will choke on excessive amount of small updates.

Incremental recovery

During recovery it is difficult to avoid having to load the image. There are, however, different methods used in page-based system for delaying loading of pages in order to minimize the downtime, thus enabling transaction processing to be started before the entire database has been restored. This requires some way of detecting references to specific pages, namely the not yet loaded ones that reside on disk. It is really a paging system that in some way bypasses or duplicates the work done by a virtual memory manager. In WS-Iris this can not conveniently be implemented since WS-Iris totally lacks an intermediate level that could catch these references.

Operating System Support

In most modern implementations of Unix, there are some way to map files into memory and this could be used as a basis for an incremental recovery, letting the operating system load the parts of the file when they are requested. It is also often possible to share memory between different processes.

Incremental Recovery in WS-Iris?

Incremental recovery are of high importance when dealing with real-time database systems. In WS-Iris, however, implementing such a schema, would require an amount of work that possibly involve rewriting a considerable amount of code concerning the internal workings in the database manager. Because of the fact that the internal *engine* implementation code is not available to the research group, this would mean that the entire WS-Iris engine would have to be rewritten from scratch.

4.2.2 Logging

Logs contain information that can be applied to a stored image bringing the database to the state it had at the same time as the log information was recorded.

There are no special constraints on the physical representation of the log, thought there are benefits, e.g. for educational purposes, if the log is man-readable. If data is to be understandable specially encoded information should also be written more human friendly. This can for example be achieved by allowing textual comments.

The size of the log is significant, since, at a high update rate, a huge log quickly will be generated. The size, however, could be significantly compressed since some of the data items stored in it are repeated multiple times.

4.3 Conclusions

The major aspects of programming, when extending WS-Iris, are those of functionality, readability, and simplicity. A highly-optimized system is of less importance. However, considered optimizations will be pointed out, thereby allowing extensions for efficiency.

The conclusions for implementation are those of simplicity and portability, whilst the use of specific methods for memory mapped files or process shared memory would at this time not be a portable solution and therefore involve too much work. Therefore neither of incremental updating nor incremental recovery are implemented.

Concerning the log materialization, readability and the size criteria are somewhat *mutually exclusive*. It would be preferable to use some well-known text compressing algorithm/program as a filter in order to be portable and readable. Compressing is more of an optimizing matter than of a functional matter.

CHAPTER 5

Implementation

In this chapter the implementation of the transaction logging and recovery system is described in detail. The first parts contains information directly concerned with the transaction logging and image handling and recovery. The last part contains some information about extensions of WS-IRIS found necessary for implementation of recovery.

5.1 Symbols

Throughout this chapter some symbols will be used for visualizing the algorithms, hopefully giving a better understanding of the possibilities of the implementation using different options. The vertical axis is a time-axis thus giving us the interpretation that the a imagined horizontal line represents the same moment and a line below some other line represents a later moment.

FIGURE 1 A vertical line represent execution of a process. An arc bursting out on either side represent a process creation (using unix's fork system call for example).

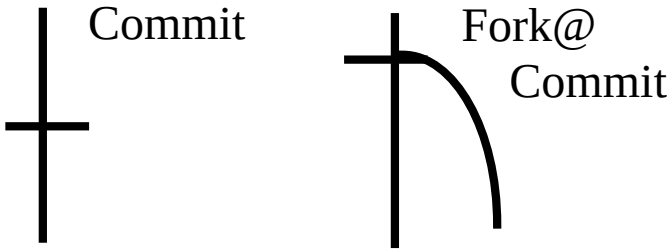

FIGURE 2 A short horizontal line represents Commit. This can be combined with other symbols as in the figure on the right where a fork is done at commit-time.

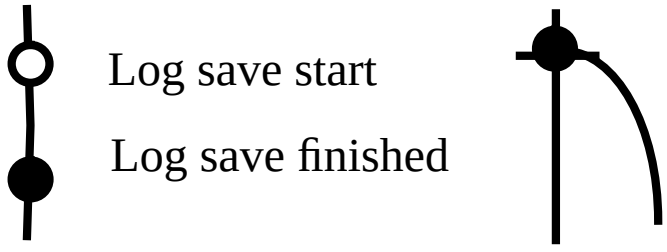

FIGURE 3 A circle represents the start of a log flush to stable storage (usually disk), and a filled ditto represents that all of the log has been flushed. On the right side Fork and Commit occur at the same time as a log flush (both start&finish). This means that at commit the log is flushed and then a fork is done.

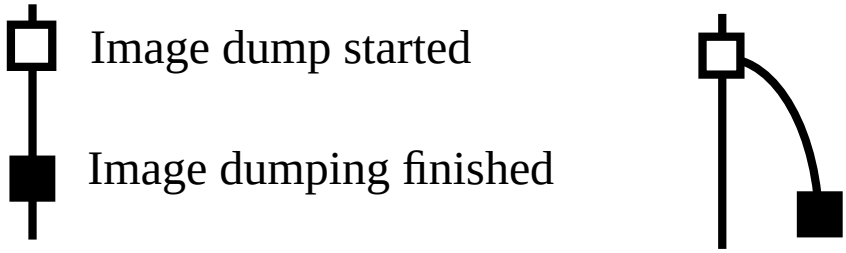

FIGURE 4 A white box represents the start of saving the image (a snapshot), and a filled box indicates that the saving operation is completed. In the right figure a process is forked. The process can be said to contain a snapshot of the parent process at the time of the fork. The child (forked) process saves the image and terminates when finished.

FIGURE 5 An arrow in direction towards an process represents a signal delivered to the process. On the right a forked child informs the parent of something using a signal.

5.2 Building a Recovery System

Using the above defined graphical language, the implemented system is explained. To begin with, simple strategies are explained leading, in the end, to the full implementation containing all optional features.

The simplest solution, i.e. not using any stored image at all, goes to the extreme of having a log only, ranging from the start to the current moment with every change of the database is saved into it. During the commit-phase redo-information is flushed onto a log-file. This is illustrated in Figure 6. The drawback with this method is that, at recovery-time, every action, from the beginning of the log has to be redone. Time used for recovery is proportional to the log size.

The other extreme is to save a snapshot into an image on disk. This requires time proportional to the size of the image. Recovery is as fast as loading the image, the used time is in some sense invariant. Since the saving of the image requires a certain amount of time this would in an ordinary system stop processing. If we, however, could do this in parallel we would gain in availability of our system, only recovery time would remain. This is the current method used in WS-IRIS, but it is in no way automatic: at commit the user gives an extra argument that tells the system to fork a child process that saves the image onto a file, a sort of no-wait save-image. These two variants are also shown in Figure 6.

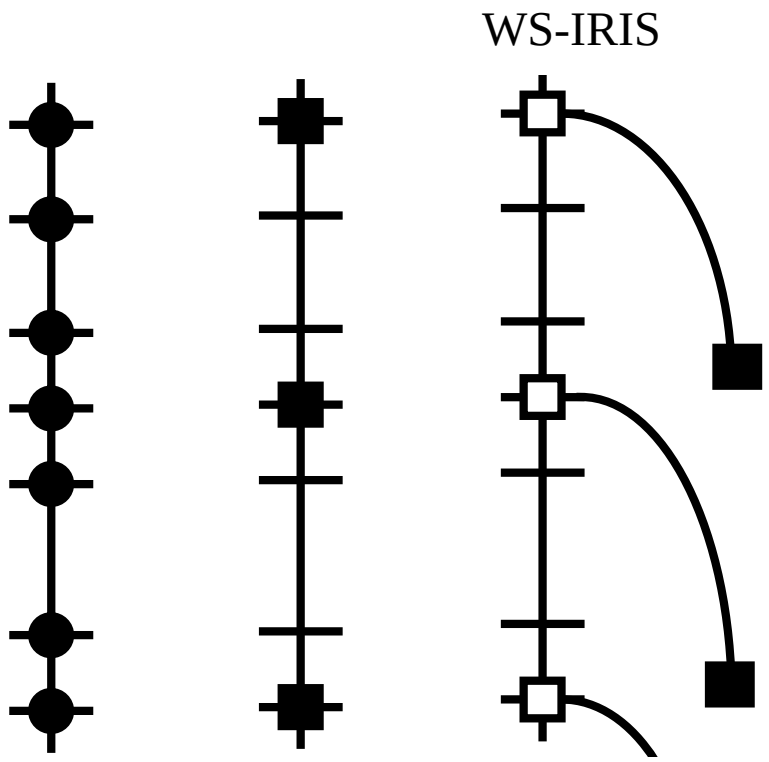

FIGURE 6

The left figure shows logging to the extreme, the middle uses only image-saving. The third uses save in a forked snapshot of the process.

This is, however, not a satisfying solution since we cannot allow committed transactions to be lost, as would be the case in these solutions. Both strategies have to be combined. The easiest solution is to flush the log at each commit and also check at commit time if save-

image has completed and if so fork a new save-image. Semaphores are used for communication and synchronisation. Figure 7 shows the solution using the defined symbols.

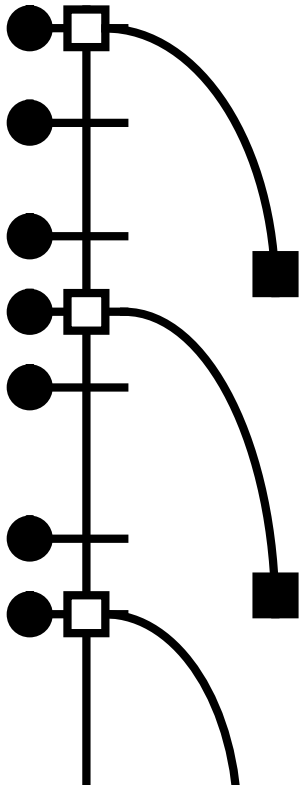

FIGURE 7 This figure combines logging on commit and fork on save-image.

This solution satisfies the transaction persistency criteria. Recovery would be rather easy using the latest fully completed saved image and the transaction log for the transactions that occurred after the image was saved.

There are, arguably, certain weaknesses. First, if the transaction commitment frequently occurred in the same interval as in which the image is saved, we would save a new image at each commit. This would lead to a constant load on disk I/O, even though we hardly update the information. Second, if there are a big number of transactions during the time used for saving one image, very much time would be spent on waiting for the redo-information (log) to be

flushed onto disk. This would compete with the current ongoing image-writing process for the sequential disk-writes.

Therefore different actions can be taken at commit-time. These are mainly concerned with blocking (waiting) and non-blocking (non-waiting) for completion of either image-save operation or log-save operation.

Different actions

a) Flush all pending log data (wait log-save).

b) Save image (wait image-save).

c) Save image in a background process (non-wait image-save).

d) Background delayed log-save.

All combinations of the four options can be set at different commit points, some, however, are only interesting at termination. They are executed in the same order as listed above. Noticeable is that *a)* guarantees persistency, *b)* selects fast recovery point, *c)* ensures overall decent recovery, and d) enables a very fast non-blocking execution, but since the log is not flushed some of the latest transaction can be lost.

Interesting combinations are

a+b: gives us persistence and a fast recovery point but is very slow at commit.

a+c: gives us persistence and overall decent recovery.

c+d: gives us a higher transaction rate, but with a less strict persistency fulfilment.

c: gives us a high transaction rate, but sacrifices on the persistency.

a+b: should be used at termination of system execution.

5.3 Language

In implementing algorithms and developing a system there are clearly benefits gained by using an interpreted language: debugging is more interactive, code can easily be changed, variables can easily be written out. All this without having to write specific code for debugging.

5.3.1 Datatypes - Lisp

The case in WS-IRIS is that most datatypes are easily accessible from within the Lisp, whilst, when using some ordinary imperative language, such as C, the source code structure would suffer from using datatypes not created for use from C.

5.3.2 C language - Operating system interface

The language C, however, is a perfect language for interfacing of operating system functions. C is also a good alternative when speed is required.

5.4 Storage-system

The storage-system is transparent to the user once it is set up. It is responsible for automatic recovery when the system is started.

The storage-system is either started manually from inside WS-IRIS by the user, or automatically when WS-IRIS is started. In the first case the user initiates the storage-system, in the other it is started indirectly by a stored image. The storage-system-code is partially part of the image in the database, and is therefore capable of recovering itself!

The transaction logging system is built up from two logically distinct parts, namely the image-handling part and the log-handling part. These parts are glued together to a powerful storage-system. On this level the concern is on the total system and it’s workings.

The steps that is used for the start of the storage-system is as follows:

1. Store parameters
2. Load image
3. Disable storage-system
4. Image recover using logs
5. Set up the storage-system
6. Restart the processes

The different parts are discussed in detail below.

1) Store parameters

This stage is only relevant when the user initializes the storage-system with new parameters or a new image. It stores the parameters concerning the recovery system outside the database image since the current image is removed when a new image is loaded.

2) Load image

If the default image exists, then it is loaded. This name is a symbolic link to the latest saved image. A rollback is done on the last transaction that was not completed when the image was saved. This transaction, if it was committed, resides in the first log.

3) Disable storage-system

The storage-system is then temporarily disabled, since the applicance of the log should not save images or create a log.

4) Image recover using logs

The two logs are loaded and applied in the appropriate order. The first log contains all transactions that where executed after the point where the save of the loaded image was started. The other log exists only if an image-save was started but not correctly completed.

Now we are back to the same state the database had just before the process was terminated, in the respect of committed transactions.

5) Set up the storage-system

The storage-system is now being set up. Either the parameters given by the user are used, if present, or the parameters that resides in the loaded image. With respect to the storage-system, it is working at the same place it was left.

6) Restart the processes

A list of actions can be performed automatically after the database has been restored. It could, in the case of a distributed database environment, be reconnecting to, or restarting other processes.

5.5 Image

For simplicity and portability (see CHAPTER 4), the Ping-Pong scheme has been chosen. Improvements done are those of asynchronous saving of the image.

5.5.1 Improvements

Improvements on the traditional implementation of the Ping-Pong strategy eliminates the fact that the database process is busy while scanning the database for writing. The time used for writing is proportional to the size of the database. This is a serious drawback, but can on most modern system be efficiently eliminated by using *Copy-On-Write* on a spawned process that shares the memory used by the first database process.

Advantage

Transaction processing can be resumed directly after the fork command, and normally, when not to many changes are scattered around to much, the overhead would be negligible.

Disadvantage

The copy-on-write method may however be somewhat insecure, because there is always the possibility that, in the worst case, the whole of the memory of the process would be modified and thereby also copied. Safety could be improved by tuning the machine's virtual memory capacity.

Another possible problem can occur due to the automatic growing image. In WS-IRIS the image grows by 25% when the image is full. It uses the Unix system call *realloc*, and no guarantees are made concerning the position of the allocated memory. So it could move around causing free blocks that are of no usage, and possible wasting of virtual memory. Also, if a growth of an image occurs while there is a forked process, there might be two different sized images that has to reside the image and start a new one, relying on the recovery system logging.

5.5.2 Algorithm

The algorithm could be seen as an interleaved process where the numbered stages occur in the given order interleaved with normal database processing. The main idea is to save the image by creating a background process at commit-time. Synchronization is implemented using semaphores, these prohibit several ongoing image-saving processes. The ping-pong schema writes to two alternative images, one at a time. When an image has been saved successfully a symbolic link in the filesystem is updated to point out the last correctly saved image.

The schema is quite straightforward:

1. Want to save
2. Synchronize — Wait for pending children
3. Initiate a new image-version
4. Fork — create a new child
5. Dump snapshot
6. Exit

The different stages are explained more in detail below.

1) Want to save

At this stage we register a need of saving, a flag is set indicating the demand of save. Then we enter the synchronisation stage at commit-time. The flag can be set at any time.

2) Synchronize — Wait for pending children

Since quite a lot of the pages might have been copied, it is desirable not to start another forked process before the previous one has exited. If we allowed several children they would be competing for the sequential disk write operation. Therefore we have to delay the fork operation.

We await the status from the child that is given at its termination. This status ensures that there is no forked child about to save the image and we can therefore start a new child.

3) Initiate a new image-version

Change variables containing information about the generation of the image.

4) Fork — Create child

A snapshot should be made consistent. This is in WS-IRIS secured by means of image-loading, since the load command does a Rollback after a successful load operation. So it is not required that fork (a snapshot) is made at commit-time since it is cleaned up later.

The process is virtually copied (descriptors for memory is copied, on write the page is copied) [Sun92Man]. This is a quite fast operation. The child process could be said to be a snapshot of the parent process.

Meanwhile transaction processing continues in the parent process.

5) Dump snapshot

The child copies the image (snapshot) onto disk. After the image is successfully written, the link to the snapshot to be used for recovery is updated.

6) Exit

The child exits. At this point the child process could be the owner of all the pages that originally belonged to the parent if the parent process has been updating data covering all pages, these pages are now freed. The parent is informed with semaphores that the process has finished.

Note: There is actually one more possibility that all pages belong to the child, this can occur when the data-image has been resized. The resizing process is not always efficiently implemented on all systems. For example, realloc on Sun always moves memory but on HP-UX it is only expanded if possible.

5.5.3 Options

Using the flag mentioned above under *1)*, one can register a demand at any time. This is automatically used if one sets the time-interval, then this flag will be set with the given interval, and automatic image-saving will occur. Other triggers can easily be implemented to register the demand of image-save.

There is an option: not letting the database processing continue before all of the image has been successfully saved onto disk. This is

achieved by giving a special parameter to the COMMIT command indicating that it should ‘flush’ it all to disk and wait for completion.

5.6 Logging

The implementation of logging also uses a relatively straightforward algorithm.

5.6.1 Improvements

Improvements done concern searching in the log and an optional asynchronously save operation.

Advantage

Most implementations studied for this report have one log, which makes it necessary to search for the transactions that are to be rolled-forward (redone). In WS-IRIS there is only one on-going transaction at a given moment, thus all of the transactions committed after a given moment have to be redone, and only one transaction has to be undone, namely the one not completed. This can however be totally ignored in WS-IRIS since the rollin operation automatically does a rollback of any uncommitted transaction.

As an option, the saving of the log can be done in a forked process, much in the same way as for the image.

Disadvantage

The worst case here is when we save a snapshot of an image after a transaction has been started. This uncompleted transaction will definitely be undone and then at application of the log, it will be redone again together with all succeeding committed transactions.

Again, when using the fork option the image could grow very large. This has to be tuned with the machine setup.

5.6.2 Algorithm

The outline of the automatic algorithm is very much the same as for Image saving:

1. Collect Commit Redo/Undo data

2. Synchronize — Wait for pending children/data.
3. Fork— Create Child.
4. Flush the log
5. Exit

1) Collect Commit Redo/Undo data.

At this stage we collect data at commit-time. This data serves as an indication of the need to save. The information stored onto the queue is redo and undo information. Then we enter the next stage.

2) Synchronize — Wait for pending children/data.

If we use the optional asynchronous approach, we have to wait for termination of any pending child that writes log information. The reason for this is that we might only handle partial data belonging to the transaction, and this data has to be written in sequence onto the same file.

The synchronisation uses to a large extent the same approach as that of fork-image-save-synchronisation. But there is one important difference that holds for the log information, and that is that it should be flushed as soon as possible onto stable storage (i.e. disk) in order to maintain the property of persistence.

3) Fork— Create Child.

This stage is only interesting in the asynchronous approach.

Since we let the child take care of all of the collected log data we can clear the internal log and continue directly with transaction processing.

4) Flush the Log.

The process (child in the asynchronous approach) writes the data sequentially onto a disk file. After the data has been flushed we can clear the log. This is not done by the child since the data in the child process is no longer of any use.

5) Exit

The child informs the parent process on exit by using a semaphore, indicating that the write operation is complete.

At this point we could, as for the image-saving process, be the only holder of all the original data pages for the image, but this again requires a tuning of the machines virtual memory.

5.6.3 Options

The main option is that of the size of the log. By initializing the system giving the maximum size of the log-file, the system automatically saves a new snapshot when it reach the limit, thus enabling us to cut the recovery time. The size must, however, be tuned against the time used for an image-recovery and the loss of time for saving the new image.

One can use an special option to let the saving of the log-information to be done in a background process. This enables a fast non-blocking transactions system for long transactions. This also, regretfully, means, that after the commit statement returned, we can not be sure that it has been successfully logged onto disk, but we know that we do not stop transaction processing. In the opposite case, when we have many but small transactions, we could also benefit from using this option

5.7 Recovery

Recovery is essentially about trying to restore the database to a well-defined state where all committed transactions have been applied.

Usually when doing recovery, one loads the latest image and searches the logfile for the position equal in time for the start of save of the last completely saved image. Then one scans forward redoing the committed transactions and undoing the non-committed transactions from the log-file.

5.7.1 Improvements

Having only one file for the log seems to be the main idea in most papers, usually one says something like ‘searching a suffix of the log’. This is however mostly an abstract way of reasoning. It is not a good way of implementing, the scheme that I have devised is to start one new physical log-file at each image-save.

Advantage

By using two log files we always know which one to load first and which one to eventually load second. The first one is the one started at the same time as the last successfully saved image was started. This log reflects all transactions after the point of starting the image saving. The second log-file exists only if a new image-save was initiated but not yet successfully completed. After a new image has been saved it is marked as the *current* image and its log-file is made the first to redo. No second log-file exists at this moment. This means that we do not have to search for the start of the log in order to do a redo operation.

Disadvantage

The use of the ping-pong checkpointing scheme uses two log-files and two image-files. The way used to indicate what is the last successfully saved images and which log-files to load is depending on the creation of links to the appropriate files. This can cause trouble since these operations (unlink/link) used in a group can not in an easy way be made atomic.

5.7.2 Algorithm for handling files

The algorithm has two stages, these are put as a wrapper around the saving of the new image. The first is done directly before starting the saving of a new image, and the second stage directly after the save operation successfully has been completed.

First stage

A new image-name is calculated. If last image was saved with the ‘ping’ suffix the new is to be saved with the ‘pong’ suffix, or the other way around. If a log-file exists with the name of the new image file with the ‘log’ suffix then it is old and therefore deleted. A symbolic link is set up so that the file becomes ‘amos.log.second’.

Inbetween stages

After the first stage the new image is saved, and directly after that on successful operation the second stage is entered.

Second stage

Symbolic links are deleted (’amos.dmp’, ‘amos.log.first’, ‘amos.log.-second’) and a new link (’amos.dmp’) is set up for the new image and one link (’amos.log.first’) for the corresponding log-file.

5.7.3 Algorithm for Recovery

The implemented recovery algorithm is quite simple.

Load Image

Load the default image-dump file, i.e. load the file that the symbolic link points to. After loading, the control is dispatched automatically to a continued recovery, since this code resides inside the loaded image.

Rollback

Since an image can be saved in the middle of a transaction, one would normally need to do a rollback of the uncommitted transactions. However, this is already done by the WS-Iris rollin command. The transaction concerned has eventually later been committed (after the image-save was started) and is then in the log.

Apply Log

Load and apply the first log-file, and then the second log-file if it exists. Application of the log is done on per transaction basis, which means that a transaction is first read in and if an endmark for that transaction is found, then applied. If an error occurs during the rollback that transaction is aborted.

Restart of Recovery System

After the database has been recovered, the recovery system is automatically restarted.

5.7.4 Lisp Extensions

To be able to handle the data and processes, extensions to the lisp had to be made in various areas. Most of the extensions are general and implement interface to operating system functionality. The different

extensions have been written as stand-alone packages that can be used both from lisp and lisp-c.

Log/History

In order to write the history events onto a file/stream, a new function that quotes strings, had to be programmed, since the existing could not handle writing to a stream using the princflag.

Processes

The original rudimentary processhandling WS-IRIS has been extended. Primitives for creating processes and giving information have been added. None information whatsoever about success on forking or execution was given. The new process handling has primitives that very much resembles the Unix processhandling primitives, but it tries to add some level of abstraction. There are functions for asking for process identifiers (pid) and waiting for a process termination in either wait or non-wait mode, asking status of active or finished processes.

Also, in this package, functions for signalling between processes can be found. One can register callback functions in lisp that are to be called, when safe, at a certain interrupt (Unix-signal). Signals can be sent to other processes. A queue of interrupts is managed and they are processed in the order they occurred.

The old interrupt catching system in WS-IRIS has been rewritten to use this package. This gives more flexibility.

Semaphores

For handling resources and for synchronising the different processes a semaphore package has been implemented. The semaphore allocation and initialization is done in a high level lisp-function. Functions are available for asking for the value of a semaphore or number of processes waiting or signalling.

Time

A mark is written in the log at the start and at the end of each transaction. The mark includes a GMT-time encoded as an integer. The start and end marks have to match for each transaction. This package has

functions for asking the GMT value from the operating system and also for converting it into a readable string for the current time-zone.

Arrays

An arraytype in WS-IRIS is normally not printed, only the id of the array is printed. In the history-log arrays occur. The lisp system must therefore be able to write and read arrays in a character-readable format. This is accomplished by adding a new print-function for the array-type. The value is written in a way that easily can be read into the lisp-system using the ordinary reader.

OID

The WS-IRIS system can write *Object IDentity numbers* (OID), but lacks support for reading them. A simple reader has been implemented that reads an OID and returns the real object. It is patched into the system.

Image-handling

The WS-IRIS system has functions for saving and loading images, but it lacked support for automatic restart. Functions has been added that are called after an image has been successfully loaded. These lisp-functions resides in the image. Extra functions can dynamically be added.

C-Items

C-Items is only two small functions, one that stores lisp-values in c-variables, and one that returns them. This is used to temporarily store lisp-data (recovery parameters) at the start of recovery so that they are available after the rollin operation, which disposes of the current image and thereby also all lisp-variables.

Trace

A simple function tracer package was written for debugging purposes. Function names, parameters and results are displayed in a structured manner.

CHAPTER 6

Evaluation

This chapter contains a short study of the resulting work and some suggestions for possible improvements and changes to the storage-system.

Tests have been performed on ‘clean’ workstations, where no other user was active, in order to minimize virtual memory usage and thereby eliminating swapping, so that WS-IRIS could benefit from the main memory residency fact.

6.1 Testing method

The tests have been performed using a modified version of a program written by Joakim Näs in his master thesis [Näs-93]. The program uses random generators, but with the same seed all the time, thereby making reproduction possible. Each run creates the same type of process and they can therefore be compared.

The program simulates a simple database containing some objects, such as persons, students and employees. First an initial set of employees and students are created. Then the ‘simulation’ starts, adding data to a stored function that contains courses taken by the students. Occasionally new students are added.

The figures shown contain only an approximated average value. What is interesting is that the values between different runs with the

same configuration yields almost the same value, variations are about 2 on 600, depending on the load on that computer.

The parameters have been varied in order to identify how much overhead the different options incur. The identified parts that are of interest are:

- cpu-time
- time used for image-storage in background
- time used for log-storage at each commit
- recovery-time

6.2 The tests

Each run does 10000 subtransactions, generating exactly 965 commits! A log that would hold all of these generated updates would have a size of 2MB. The image created is about 2MB.

Test	Time[s]	Main[%]	Child[%]
No disk-I/O	590	90	
Storage Image&Log -> 18#	870	60-40	1-5
Storage Log only -> 2MB	760	70-80	
Recovery Log (2MB)	214	80-90	
Storage Image -> 34#	729	90-80	1-5
Storage Image, 2 mins -> 6#	610	90	1-5

TABLE 1 This table contains runs from a Sun-4 ELC workstation.

Test	Time[s]	Main[%]	Child[%]
No disk-I/O	204	99	
Storage Image&Log -> 16#	340	60-80	1-4
Storage Log only -> 2MB	304	70-80	
Recovery Log (2MB)	74	90	
Storage Image -> 15#	230	90	1-4
Storage Image, 2 mins -> 2#	210	90-98	1-5

TABLE 2 This table contains runs from an HP9000 workstation.

In Table 1 the results are shown for a Sun-4 ELC workstation, and in Table 2 the results for HP9000 workstation. Numbers in the *Test* column in the tables followed by an hash-symbol (#) are number of images saved. The *Main* and *Child* shows % CPU-usage.

From the tables one can see that the HP workstation is significantly faster than the Sun workstation. The HP therefore saves fewer images, since it executed faster. In Table 3 the derived information is shown.

Factor	Sun [s]	HP [s]	% CPU
Internal-processing	590	204	90—99
Log-saving	170	100	
Images 34@Sun, 15@HP	139	26	1—5
Images, 6@Sun, 2@HP	20	6	1—5
AMOS overhead, 1 image	4	2	
Recovery, 2MB	9 KB/s	27 KB/s	90 %

TABLE 3 Calculated information from Table 1 and Table 2.

From this an approximate formula may be set up:

$$T_{\mathbf{Internal}} + T_{\mathbf{Log}} + N_{\mathbf{Images}} \cdot T_{\mathbf{Image}} \approx T_{\mathbf{Storage}} \qquad \textit{(EQ 1)}$$

$$\text{(Sun:) } 590 + 170 + 34 \cdot 4 = 896\ (870) \qquad \textit{(EQ 2)}$$

$$\text{(HP:) } 204 + 100 + 15 \cdot 2 = 334\ (340), \qquad \textit{(EQ 3)}$$

where $T_{\mathbf{Internal}}$ is time used for internal processing in WS-IRIS, $T_{\mathbf{Log}}$ is the time overhead used for logging onto disk, $N_{\mathbf{Images}}$ is the number of images that were written onto disk, and $T_{\mathbf{Image}}$ is the time overhead for saving an image. $T_{\mathbf{Storage}}$ is the time for the total processing depending on the variables.

The values in parentheses are the expected (actual) values of $T_{\mathbf{Storage}}$. These can be found in table 1 and table 2 on the *Storage Image&Log* row. The calculated values differ +3% (HP) and -2% (Sun) from the expected values. A formula for the Recovery time ($T_{\mathbf{Recover}}$), where

$S_{\mathbf{Log}}$ is the size of the log, $V_{\mathbf{Log}}$ is the speed that the log is written onto the disk, and $T_{\mathbf{Load}}$ is the time to load an image, can now be written:

$$T_{\mathbf{Recover}} \approx T_{\mathbf{Load}} + S_{\mathbf{Log}} / V_{\mathbf{Log}} \quad \textit{(EQ 4)}$$

From this the size of the log may be calculated:

$$S_{\mathbf{Log}} \approx V_{\mathbf{Log}} \cdot (T_{\mathbf{Recover}} - T_{\mathbf{Load}}). \quad \textit{(EQ 5)}$$

This gives a function that gives the size of the log for a certain recovery time. By tuning the size of the log the recovery time can be optimized (zero log gives minimal recovery time). But that would require the system to continuously save a new image as often as possible. That would not be particularly efficient since there would be a constant overhead. Therefore, it is important to choose the maximal log size not to make too much overhead at recovery time. Actually the time used to load and apply the log should be less than the time used for loading an image.

Setting $S_{\mathbf{Log}} / V_{\mathbf{Log}}$ less than $T_{\mathbf{Load}}$ and having that $T_{\mathbf{Load}} = S_{\mathbf{Image}} / V_{\mathbf{Image}}$, where $S_{\mathbf{Image}}$ is the size of the image and $V_{\mathbf{Image}}$ is the speed of loading the image, the result is:

$$S_{\mathbf{Image}} < S_{\mathbf{Image}} \cdot V_{\mathbf{Log}} / V_{\mathbf{Image}}. \quad \textit{(EQ 6)}$$

Since image processing requires less overhead it is natural to assume that $V_{\mathbf{Image}} > V_{\mathbf{Log}}$ giving:

$$S_{\mathbf{Log}} << S_{\mathbf{Image}}. \quad \textit{(EQ 7)}$$

A practical *rule of the thumb* is to set the maximum log size to be a tenth of the size of the image.

6.3 Possible Improvements

Image handling

Possible problems concerning the image is the growth as mentioned in section 5.5.1 Improvements.

One possible solution is to wait for the forked process to finish before reallocating. Another solution is to allocate in advance so that the image is big-enough, in reality having a statically allocated image. A third more interesting strategy would be to kill the process that saves

the image and start a new one, relying on the recovery system logging.

Compressing Log Data

Studying the log file, one realizes that there are duplicates of data written on to the log. The log is written in plain text and therefore does not contain any identity of lists, strings or arrays. These are allocated each time they are used, thereby introducing some storage overhead in the database. One could check, when allocating a new string if this string already exists, and if so reuse it. This assumes that the strings cannot be changed. This would be very much like symbols reusage and identity. But as with strings, lists and arrays have their identity in what they store and the checks for finding a possible identical list or array (as in lisp equal) would have to traverse all of its elements.

More promising is to use coding compression, that is, a more compact storage format than the textual representation of the data.

Removing Undo Information

The log currently contains both redo and undo data. The loading time could possibly be reduced somewhat by removing the undo data. The biggest gain would be not having to allocate storage space for the data structures that are stored in the undo data since this data is not used and thereby soon deallocated. The removal of the undo information would currently be legal since only committed transactions are written out to the log. This would however not be true if log-operations were stored before they had been committed.

Configuration

The configuration process of the recovery system could be programmed to be adaptable and dynamical. When there is mostly long transactions log could be saved in a background process, when there are many small operations they could be grouped together and be taken care of as if they were one long transaction. In both cases minimizing the write overhead in the current process. The functionality exists for background log saving, but setting these options can be hard even for an expert, since information must be known about the database usage and application and most likely the system should be adoptable to the current system and transaction load.

6.4 Software Engineering aspects

In order to do this work, information had to be gathered about the WS-Iris system and a number of research reports that describes the system and its ancestors had to be read. Chats with the creator Tore Risch has been enriching not only concerning the WS-IRIS system, but also implementation details. The WS-IRIS system lacks collected documentation about the internal workings. Bits and pieces are spread over many research reports and documentation papers.

However, since WS-Iris was prototyped in Lisp and later migrated towards an implementation in C for efficiency, parts in Lisp remains. The reason for not migrating those parts is mainly because of the small benefits that thereby would have been gained. Namely, these lisp functions resides in the database-image as lisp-objects. Using the pretty-printer, that every serious lisp interpreter should have, calls, internal functionality and implementation could be studied.

By using sophisticated research methods (trial-and-error), knowledge has been gathered about the necessary implementation details.

During this time many papers has been read about similar systems that implements recovery. Different strategies has been studied in the search for ideas for a recovery system for WS-IRIS.

The requirements were identified, and high level support for operating systems functions was implemented. Experiments on image-handling were done, thereby building up a safe system for handling processes for background operations. Functions for handling the writing and reading of log-files required changes in low-level functions for WS-Iris internal datatypes. Some bugs were found in the process and these have been corrected.

Using these image- and log-handling routines a recovery system was implemented, with support for fully automatic recovery at start-up.

6.5 Acknowledgments

Work has been inspiring, thanks to the support from the research group CAElab and Tore Risch. Insight in the database community and other research areas has been enriching. Also many thanks to Henrik Nilsson who has read and commented this emerging report.

CHAPTER 7

Definitions

This chapter contains explanations of common words and concepts within OODB.

Object Orientation (OO)

From OO one should expect some sort of inheritence and encapsulations of data and methods.

Object Oriented DataBase (OODB)

It is still in the definition stage, there is no distinct definition but one should expect fullfilment of Object Orientation (OO).

Object Oriented Programming (OOP)

Programming using the concepts of Object encapsulation and well-defined methods on Objects Data. It is a programming style, not a requirement of the language, but it helps.

Transaction

Changes such as adding or deleting data in a database. In a transactional system one executes Commit and when one regains control one knows that ”to some extent” the changes are permanent and accepted.

Log

A log can be described as a infinite sequence, in one direction, of log records that describes changes in the state of a database. Often stored on a file.Contains also information about committed transactions.

Log Record

Contains information about where and what the changes were.

History

A history in WS-Iris is a list of atomic subactions within a transaction. ROLLBACK undoes changes using history. COMMIT clears histories.

ROLLBACK

Restores the state of the database to the state it had before the transaction was initiated, some sort of abort/undo. Histories and their undo(old) information is used.

COMMIT

Executing COMMIT in the database ensures that the information is permanently stored and can not be ROLLBACKed (undone). If transaction logging is used the loggs new information is secured in some way.

Image

The data area called ”image” in the WS-IRiS system contains all of the database and interfacing information for the database implementation.

Dump

An snapshot stored on disk is called an dump. A dump usually also has some ”relocation” information.

Snapshot

A snapshot is said to be taken when a image is copied as an atomic operation. Compare to the Unix fork operation.

Consistent

When a database is in a well defined condition it is said to be consistent. There are no incomplete transactions.

Persistent

”Is stored.” That is it is permanently stored, changes are stored inbetween successive runs of a program.

Consistent Image

An image with only commited data. And thereby consistent.

CheckPointing

Checkpointing is the collection and storing of information in a ”safe” way. It is used for limiting the number of operations that has to be REDOne after a crash.

Compare ”CheckPoint” in the military sense. CheckPoint Charlie in Berlin was an example of a famous such.

CHAPTER 8

References

[Litwin-92] Witold Litwin, Tore Risch (1992), Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, IEEE Transactions on Knowledge and Data Engineering, Vol 4, No 6, pp. 517--528.

[Fishman-87] D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, W.K. Wilkinson (1987), Object-Oriented Concepts, Databases, and Applications, Chapter 10: Overview of the Iris DBMS, pp. 219--250.

[Risch-92]Tore Risch (1992), WS-IRIS; Memory Object Oriented DBMS, Technical Report HPL-DTD-92-5 April 29 1992, Database Technology Department, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304.

[Garcia-Molina-92] Hector Garcia-Molina (1992), Kenneth Salem, Main Memory Database Systems: An Overview, IEEE Transactions on Knowledge and Data Engineering, Vol 4, No. 6, pp. 509--516.

[Haerder-83]Theo Haerder, Andreas Reuter (1983), Principles of Transaction-Oriented Database Recovery, ACM, Transaction on, pp. 151--166.

[Salem-90]Kenneth Salem, Hector Garcia-Molina (1990), System M: A Transaction Processing Testbed for Memory Resident Data, IEEE

Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, pp. 161--172.

[Levy-92]Eliezer Levy, Avi Silberschatz (1992), Incremental Recovery in Main Memory Database Systems, IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6, pp. 529--540.

[Sun-92Mem]Sun Microsystems, System V Interprocess Communication Facilities, System Software AnswerBook Relase 1, Issue 3, Programming Utilities and Libraries, 3.5. Shared Memory, pp. 3-35--3-47.

[Sun-92Sem]Sun Microsystems, System V Interprocess Communication Facilities, System Software AnswerBook Release 1, Issue 3, Programming Utilities and Libraries, 3.4. Semaphores, pp. 3-19--3-35.

[Sun-92Man]Sun Microsystems, manpage: semget, pp. 2-173--2-174. SunMicrosystems, manpage: semop, pp. 2-175--2-177. SunMicrosystems, manpage: semctl, pp. 2-170--2-172.

[Näs-93]Joakim Näs (1993), Randomized optimization of object oriented queries in a main memory database management system, Master’s thesis, LiTH-IDA-Ex 9325 Linköping University.

Avdelning, Institution, fakultet
Division, department, faculty

Department of Computer and Information Science

Institutionen för datavetenskap

ISBN:

ISSN:

Rapportnr: LiTH-IDA-Ex9404
Report no:

Upplagans storlek:
Number of copies:

Datum:
Date:

Projekt:
Project:

Titel:
Title: En implementering av transaktionsloggning och återhämtning i ett minnesresident databassystem.
An Implementation of Transaction Logging and Recovery in a Main Memory Resident Database System.

Författare:
Author: Jonas S. Karlsson

Uppdragsgivare:
Commissioned by: Tore Risch, CAElab, LiTH-IDA.

Dnr:
Call no:

Rapporttyp:
Kind of report:
- [X] Examensarbete/Final report
- [ ] Delrapport/Process report
- [ ] Reserapport/Travel report
- [ ] Slutrapport/Final report
- [ ] Övrig rapport/Other kind of report

Rapportspråk:
Language:
- [ ] Svenska/Swedish
- [x] Engelska/English
- [ ] _____

Sammanfattning (högst 150 ord):
Abstract (150 words)

This report describes an implementation of Transaction Logging and Recovery using Unix Copy-On-Write on spawned processes. The purpose of the work is to extend WS-Iris, a research project on Object Oriented Main Memory Databases, with functionality for failure recovery.

The presented work is a Master Thesis for a student of Master of Science in Computer Science and Technology. The work has been commissioned by Tore Risch, Professor of Engineering Databases at Computer Aided Engineering laboratory (CAElab), Linköping University (LiU/LiTH), Sweden.

Nyckelord (högst 8): Transaction logging, logical logging, recovery, main memory database, copy-on-write, process forking
Keywords (8):

Bibliotekets anteckningar