J. M Bishop
24/3/91

South African Computer Journal
Number 4
March 1991

Suid-Afrikaanse Rekenaartydskrif
Nommer 4
Maart 1991

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the South African Computer Society and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Suid-Afrikaanse Rekenaarvereniging en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083

Assistant Editor: Information Systems
Dr Peter Lay
P. O. Box 2142
Windmeul 7630

Editorial Board

Professor Gerhard Barth
Director: German AI Research Institute
Postfach 2080
D-6750 Kaiserslautern
West Germany

Professor Judy Bishop
Department of Computer Science
University of the Witwatersrand
Private Bag 3
WITS 2050

Professor Donald Cowan
Department of Computing and Communications
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

Professor Jürg Gutknecht
Institut für Computersysteme
ETH
CH-8092 Zürich
Switzerland

Professor Pieter Kritzinger
Department of Computer Science
University of Cape Town
Rondebosch 7700

Professor F H Lochovsky
Computer Systems Research Institute
University of Toronto
Sanford Fleming Building
10 King's College Road
Toronto, Ontario M5S 1A4
Canada

Professor Stephen R Schach
Computer Science Department
Vanderbilt University
Box 70, Station B
Nashville, Tennessee 37235
USA

Professor Basie von Solms
Departement Rekenaarwetenskap
Rand Afrikaanse Universiteit
P.O. Box 524
Auckland Park 0010

Subscriptions

	Annual	Single copy
Southern Africa:	R32-00	R8-00
Elsewhere:	$32-00	$8-00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Computer Science Department, University of Pretoria, Pretoria 0002
Phone: (012) 420-2843 Fax: (int) + (012) + 43-6454
Sigma Press

Guest Editorial

Does Today's Industry Need Qualified Computer Scientists?

This guest editorial consists of two contrasting views on the value to industry of a professional degree in computer science. Both authors, one local and one from Germany, are managing directors of well-respected software houses. (Editor)

Viewpoint I

Hans G Steiner
MBP Software and Systems GMBH
Semerteichestrasse 47-49
D4600 Dortmund 1

I would like to begin by recounting from my student days a story that I consider to be relevant. While attending a career forum for computer scientists, mathematicians and physicists, the personnel officer from IBM Germany was asked if he would consider taking on mathematicians. The gist of his answer was as follows: "Of course I must admit that I could just as well give the mathematician's job to a theologian. What is important is the ability to think logically. It is only there, on the job, that he learns how to become productive for us."

This episode occurred 14 years ago at a time when graduating mathematicians did not necessarily learn programming and when computer scientists were few and far between. The situation has improved immensely since then. Mechanical engineers, electrical engineers and physicists, all with programming knowledge, have for the most part taken over many programming jobs. This shows industry that, as time goes by, the answer to the opening question is becoming an ever-louder and more frequent "NO".

I support this opinion and in the remainder of this essay I will expand on my reasons, as well as highlight some exceptions.

An employee who is recruited directly from a university should possess the following four capabilities:

1. An ability to think logically: One of the basic requirements in our business is the ability to recognise, analyze, structure, break down and solve a problem as well as to fully synthesize the solution. The important thing is to break down the problem in such a way that the individual components can feasibly be solved. This is what distinguishes an engineer/scientist from an arts scholar. The latter usually concentrates on the complete problem and tends to settle for a contentious, complex and partially non-feasible solution. In our business, it is not enough to merely ask the "right" questions.

This ability to think logically may be accentuated in computer science; however engineers/scientists will generally possess the ability to an equal extent.

2. Programming skills: Our employees' prime tool of the trade is their ability to encode solutions to problems. Ideally, this ability ought to be held as abstract as possible. In other words, the further away from the "bit", the better. FORTRAN programmers who, for example, concentrate on the multiple use of memory space of all variables will never be successful programmers in an object-oriented programming language.

The difference can be seen, even in today's universities. For example, one only has to read a PROLOG program from a student who learned PASCAL in his first semester and PROLOG in his fifth. On average, this is always a "PASCAL program in PROLOG". The various possibilities offered by a predicate calculus language are only recognised and used by the best students. Again, we do not need the average computer science scholar who has spent between six and eight years writing complicated PASCAL programs, but rather the "thinker" with basic programming knowledge who is capable of abstracting the task. Once again, the ability is independent of faculty.

3. Teamwork skills: Working successfully in a team

This SACJ issue is sponsored by
Department of Computer Science
Rhodes University

requires assertiveness, tolerance, stability and one's own ideas. Very few problems have solutions that can be managed by one person successfully in the allocated time. Out of 700 employees, we can only afford approximately five "lone warriors" who are, in turn, the leading specialists in a wide field. They have a strategic vision which we follow. All remaining employees are evaluated, for better or worse, on their team performance. Some people have an in-built ability to work in teams. A few universities - unfortunately not enough - encourage this team-thinking. Again we see that the ability is independent of university faculty.

4. Motivation: The ability to enjoy one's particular job is a major driving force in every employee. Whereas in the sixties everything had to be "bigger, faster and better" and in the eighties "things had to be meaningful to society", the theme for the nineties is self-realization. Those companies who succeed in incorporating different employees (ie employees with different driving forces) into the company culture and who motivate each employee optimally will be successful in the nineties and beyond. There are huge productivity gains to be had from motivating employees. Compared with this, the possibilities offered by CASE tools pale into insignificance.

One basic requirement is thus the recruitment of a self-motivated employee who should at no stage become demotivated, whether it be by company culture, superiors or working conditions.

Again, this is not linked to a specific university faculty and is independent of know-how.

As none of these four capabilities are necessarily restricted to studies in computer science, the technical/-scientific background of new employees who are being recruited is largely irrelevant.

I would now like to point out a few exceptions which might give a computer scientist the upper hand in an interview. I refer exclusively to our own company and our specific company tasks.

1. Porting our COBOL Compiler onto the latest UNIX machine from the manufacturer XY. Knowledge of the UNIX operating systems could be very valuable and enable the new employee to rapidly become productive.
2. Programming the 37th interface (special customer request) for our ISDN card. Knowledge of interface protocols or experience with protocol conversions would be very useful and could be a decisive factor. Such specialized knowledge is usually very rare.
3. Adapting our integrated office automation system to the 17th foreign language. The employee must command the language perfectly. Simply outsourcing the translation would mean that this language version could not be maintained or supported. From this example one can see that specialized knowledge not only refers to knowledge gained from computer science studies.

In the product business, it sometimes happens that computer scientists with specialized knowledge are sought. (This is almost impossible in the project business, due to the variety of tasks to be performed.) However such a "knowledge" advantage over others usually only lasts about a year. After that, the achievements of two different employees (one with specialized knowledge and the other without) tends to even out.

Most applicants who start out do not know our products, as the flow of employees in this industry is almost always from manufacturer to user. Hardware and software manufacturers often lose their products specialist to the products' users. Seldom do employees change in the other direction.

In my opinion, universities can learn two things from this essay:

1. Studies in computer science give basic knowledge that can be used in various jobs. The student should however be careful not to place all his eggs in one basket.
2. Teamwork should be encouraged more. Time allows for very few geniuses, acting as 'lone warriors', to initiate progress in our society.

I have taken the liberty of basing my interpretation and answer to the opening question on my own judgement and experiences. I would be grateful for other opinions and experiences on this topic.

I would like to conclude by expressing my gratitude for having had this opportunity to express my views.

Viewpoint II

Pierre Visser
Grinaker Informatics, P.O.Box 29818,
Sunnyside, 0132

The title question currently generates as many viewpoints as a counterpart question: "What is the correct curriculum for a computer science qualification?" Such questions stem from the many and diverse requirements expected to be fulfilled by the still developing applied science. A basic assumption of this editorial is that we need to have an explosion and consolidation in computer science theory. Only after this has occurred will a more general consensus of opinion exist - as is the case in other matured sciences.

An argument is presented here for the current approach of striving towards a balance between immediate industry needs and long term perceived theoretical requirements of industry, even though the balance, as viewed from either side, will always be imperfect.

Industry can, of course, do without qualified computer scientists - that is how it was established. Dedicated mathematicians, physicists, engineers and other scientists will, as in the past, continue to effect improvements. However, as one of those scientists from the

early days, it is difficult for me to understand why one would choose to continue this way.

A computer science qualification is viewed here as a university education (4 years) into theory that is not obtainable otherwise. By definition, therefore, a qualified computer scientist is not trained to conform to specific job requirements. Rather, the computer scientist will possess knowledge that will serve him long past the present day's computing technology.

Whether industry needs qualified computer scientists depends on two issues. Firstly, can an education be provided for computing technology that will serve as a foundation for the student's next 45 years in industry; and secondly, can industry build upon this foundation to created wealth more effectively than without qualified computer scientists.

It is widely accepted that, in broad terms, the teaching of fundamental theory will serve the first purpose. However, what subject matter to include from the wealth of mathematics, physics, OR, and from computing fields such as networking, operating systems and others, remains the illusive issue. Universities can merely strive to select the right mix for the perceived future needs of industry. This requires insight into the evolution of computing technology. I will later discuss such insight as a basic requirement for a qualified computer scientist.

What is important in teaching is to focus on fundamental theory. Just as the natural science student needs to breed fruit flies in order to gain insight into the dynamics of inheritance, so too the computer science student needs to develop software. The purpose should be to create understanding and insight into fundamental theory, and, just as in the case of the breeder of fruit flies, the software developed should never be measured against efficiency requirements from industry.

The second issue is whether industry can build on this theoretical foundation to create wealth.

A depth of insight into computing technology, more so than with other training, can be identified as the focus of the potential value of a qualified computer scientist to industry. Three areas which require such insight are discussed below, namely organisation, product definition and the application of new computing technology in industry.

Computing products form an integral part of an organisation, and represent a significant capital investment aimed at increasing efficiency. These products are incorporated in an evolutionary way to match changing organisational requirements with improving product capabilities. Decisions to use products determine the long term efficiency and cost-effective replacement. Such decisions require insight into computing technology and its evolution. A qualified computer scientist can improve such decisions only if he gains enough insight into computing technology as well as its interaction with business through years of practice.

The success of products in some areas is dependent on market requirements which depend on computing technology and its evolution. The correct definition of characteristics of products that interface to computers is such an example. Insight into computing technology is able to create the versatility, simplicity or other improved selling features which can open new market segments.

The third area where insight into computing technology plays an important role is in the application of new computing technology (or a new trend) in an organisation. Examples include the introductory period for networking, DBMS-technology, distributed processing and document image processing. In areas such as these, the newly qualified computer scientist can be applied effectively and at the same time build up insight through experience which he will require for the other areas of organisational and product decisions mentioned above.

A major dilemma in the continuous development of insight into computing technology by qualified computer scientists is their correct application in industry. The identification of the opportunities within the three areas discussed above, requires insight into computing technology itself. Winning companies that depend on computing technology have this ability. In such companies the insight of the qualified computer scientist into computing technology as well as its contribution to the business is constantly stimulated, turning the qualified computer scientist into a valuable company resource.

What has been neglected in this whole discussion is the role of the "technician" and of the casual user of computing technology. Such personnel are required to implement selected computing technology of the day efficiently, whether in accounting, chemical engineering or other specialised disciplines. Their role and place is unquestioned. However, it cannot be expected of them to evaluate the potential of new computing technology, formulate algorithms from fundamental theory or any such decisions which require insight built upon a sound theoretical knowledge of the field.

The final aspect in answering the opening question is whether the qualified computer scientist can outperform other professionals who build up their own experience in computing technology. Many examples could be cited of improvement brought about by non-computer scientists in the past. However, these individuals formed part of the bootstrapping for computer science theory and education. We should have faith in this bootstrapping of computer science qualifications, because computing technology will increasingly diversify into many directions of specialisation in years to come, each requiring a body of fundamental theory.

This complexity cannot be left to a casual development of insight - industry requires qualified computer scientists to experience interaction with business objectives in order to cope successfully with future computing technology.

Database Consistency under UNIX

H L Viktor and M H Rennhackkamp
Department of Computer Science, University of Stellenbosch, Stellenbosch, 7600

Abstract

A recovery techniqu. to facilitate the recovery from system failures should ensure a consistent database state at all times. The UNIX write system call uses a delayed-write policy. Data blocks are kept in a buffer cache and are later asynchronously written to disk. If a system failure occurs, logically complete transactions may be lost and the database may contain incorrect old data values. If a physical write was being executed when the failure occurred, the database will be inconsistent, since an update in place policy is used when writing to disk. Differential files can provide database consistency efficiently.
Keywords: *Consistency, recovery, system failure, UNIX.*
Computing Review Categories: D.4.3, H.2.2.

Recieved March 1990, Accepted October 1990

Introduction

The NRDNIX distributed database management system (DDBMS) [6], currently under development at the University of Stellenbosch, utilizes a redo, no-undo recovery algorithm to facilitate the recovery from system failures. In the event of a system failure, the contents of main memory are considered lost. A previous consistent database state is constructed and a logical logging facility is then used to redo transactions that have since been committed. The DDBMS is implemented on top of the UNIX operating system and accesses the physical database through standard system calls.

The UNIX *write* system call uses the delayed-write policy. To reduce the amount of I/O traffic, accessed data are kept in a buffer cache. When a *write* system call is issued, the corresponding buffer is marked accordingly. The actual propagation of the data is performed at a later stage. If a system failure occurs, there could be logically complete but physically incomplete I/O requests in the buffers. These will be destroyed, causing inconsistency between the user's view and the physical database. If a failure occurs while a physical write operation is taking place, both the old and the new data values may be lost. In some cases the determination of a previous consistent database state may be impossible.

A mechanism using tagged differential files to maintain a consistent database state, even in the event of a system failure, is described. The basic idea is to ensure the existence of a previous state at which the database was consistent. This eliminates the problem associated with the UNIX delayed-write and update-in-place policy.

In the following section a description of the UNIX file system components is given and the inconsistency problem is outlined and illustrated by means of an example. This is followed by an overview of the recovery techniques implemented in the NRDNIX DDBMS. The implementation of the differential file technique is described. The consistency achieved by the recovery processing algorithm is illustrated by means of the example. It is concluded that the proposed method provides the application with a consistent view of the database.

1. The UNIX Kernel

One of the main objectives of the designers of the UNIX operating system was to provide a small, simple yet effective, operating system. This gives users the freedom to design their own user interfaces and other utility programs as needed. The operating system consists of 10 000 lines C code and approximately 1 000 line of assembler code. The operating system is often called the kernel, emphasizing its isolation from user programs. The kernel consists of two main components: the process subsystem and the file subsystem [5].

Processes

An application program is executed in an environment called a user process. A user process consists of a text, data and stack segment and executes in user mode. Various processes are concurrently executed in an interleaved fashion

as determined by the kernel's scheduling algorithm. These processes communicate via system calls. Processes are protected from interference from others by ensuring that one process may not read or write the data or stack of another. The read-only text segment may be shared by various processes.

When a system function is required, the user process calls the system as a subroutine via one of the standard UNIX system calls. This causes a distinct switch of environment and the process enters kernel mode until the completion of the kernel subroutine. A separate stack is used for the user and kernel modes, thus isolating the user from the operating system routines.

The File System

The UNIX kernel implements a *hierarchical file system*. A file system consists of a sequence of logical blocks, each containing any convenient multiple of 512 bytes. This paper will use the term 'block' to refer to a logical block consisting of 512 bytes. Directories are used to implement the hierarchical structure. The nodes of the hierarchy are directories and the leaves are files, with the root directory as the parent node of the system. A directory is a file consisting of file names as well as information to access these files. Directories thus provide the mapping between the names of files and files themselves. Below the hierarchical file subsystem a *flat file subsystem* is implemented by means of a list of file definitions called an i-list. These file definitions, called inodes, contain the relevant file information to access and modify a file. Below this flat file system is the *I/O subsystem* which carries out physical I/O operations on devices. Below this is the *physical hardware*. Figure 1 shows the relationship between the various subsystems [1].

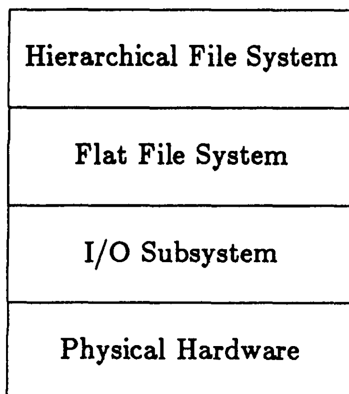

Figure 1: The various file subsystems.

A directory entry contains a file name and a pointer to the file, called the i-number of the file. The i-number is used as an index to the i-list stored in a known part of the device on which the directory resides. The inode found thereby contains a description of the file. A copy of the inode is kept on disk when the file is not active. When a file is accessed, the inode is read to main memory. The kernel uses the file descriptor to access the file descriptor table. It follows pointers to a global file table and inode entry. The physical location of the data file is found by inspecting the inode. The in-core inode is kept locked to ensure its consistency during update. The use of inodes simplifies the design and maintenance of the file system.

The I/O subsystem allows processes to communicate with peripheral devices such as disks, tapes and printers. The kernel modules that control devices are known as device drivers. An application interfaces devices through the file system, accessing the device by means of a name similar to a file name.

The actual I/O to disk is controlled by the device driver. The driver supervises the transmission of data between primary storage and disk. A disk may be partitioned into several units, each containing one or more file systems. The kernel deals on a logical level with file systems rather than with disks. The conversion between logical device addresses and physical device addresses is done by the disk driver. The disk driver translates a file system address, consisting of a logical device number and a block number, to a particular sector on disk. A disk may be accessed in either block or character mode. The standard *read* and *write* system calls use the block I/O system. Figure 2 shows the execution path of a disk access.

The Buffer Cache

The kernel attempts to minimize the frequency of disk access by keeping a pool of internal buffers, called the buffer cache, containing the recently accessed disk blocks. The cache acts as an interface between the file system and the block device driver.

When a *read* system call is executed, the kernel first attempts to read from the buffer cache. If the data are located in the cache, the kernel does not have to read from disk. If not, the disk block on which the data are located is read to the cache. The data are copied from the cache to the user address space and the corresponding cache buffer is released. The kernel locks the corresponding inode for the duration of the call. This ensures that no other process may update the block of data during the *read* system call. The kernel unlocks the inode between various system calls. If another process changes the file between two *read* system

calls, the first process may read unexpected data although the kernel data structure remains consistent. The system does not guarantee that the data in a file remains the same after the file has been opened.

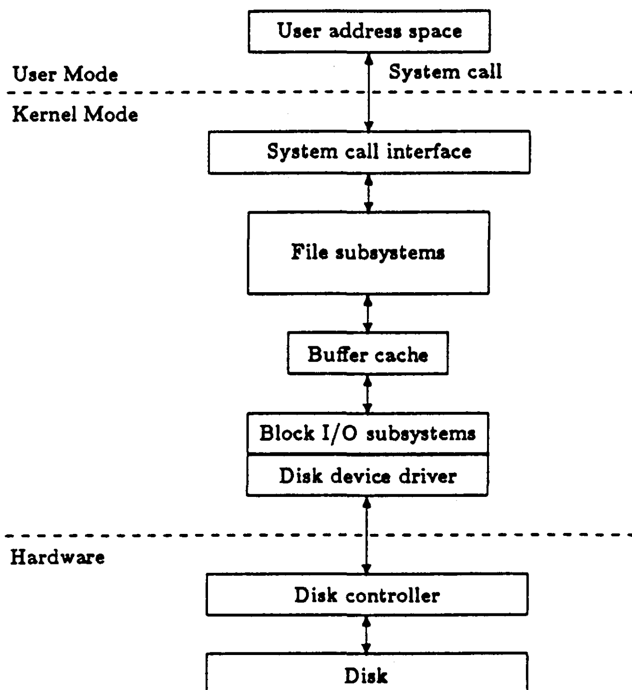

Figure 2: The execution path of a disk access.

Similarly, data written to disk are cached for enhanced retrieval time. If the kernel informs the disk driver that it has a buffer whose contents should be output, the disk driver schedules the block for I/O. When a delayed-write is executed, the buffer is marked as waiting to be written and is not immediately written to disk. If the file does not contain a block corresponding to the byte offset to be written when issuing the *write* system call, the kernel allocates a new block corresponding to the logical byte offset to be written. Blocks are iteratively written to disk. During each iteration, it is determined whether the whole block should be written to disk or only part of it. If only part of it is written, the block is first read from disk to avoid overwriting the part that will remain the same. Otherwise, the whole block is written to disk.

File System Inconsistencies

Due to the design of the I/O system, write operations could cause an inconsistent file system, resulting in loss of information.

- The asynchronous nature of the delayed-write algorithm makes error reporting and meaningful user error handling almost impossible, resulting in possible undetectable inconsistencies.
- The physical write is delayed as long as possible, saving extra I/O operations. Since the kernel does not immediately write data to disk, the system is vulnerable to crashes that leave the disk data in an incorrect state. If a system failure occurs there could be logically complete, but physically incomplete I/O in the buffers. The standard I/O packages available to C language programs include a *fflush* call. This function call flushes data from buffers in the user address space into the kernel. Periodic use of this function helps, but does not solve the problem, since the application still does not know when the kernel writes the data to the disk.
- An update-in-place policy is used when writing a block of data to disk. If a system failure occurs during this process, the resulting file can be in any of the following states:
  - The write was logically complete, but the physical write had not been initiated. This results in loss of information.
  - The write was in progress when the failure occurred, resulting in an inconsistent block of data.
  - The write was both logically and physically complete.

An application issuing a UNIX *write* system call is not notified when the data are physically written. The number of disk writes are reduced at the expense of possible loss of information in the event of a system failure.

In a database environment such as NRDNIX, operations are grouped into transactions that are concurrenctly executed on the database files. The consistency and currency of data must be ensured. This is achieved by requiring that the effects of a transaction must be atomic. They are either totally reflected in the database files or not at all. When the normal UNIX system calls are used, the atomicity cannot be guaranteed, as illustrated by means of the following example.

Example 1

A simple example is used throughout this paper. A client at a departmental store is identified by a number and is further described by age and credit balance. A bonus of R50 is given to clients older than 65, with a balance greater than R100, resulting in the following transaction:

BEGIN TRANSACTION;
UPDATE client
SET balance $=$ *balance* $+ 50$
WHERE age > 65 *AND*
balance > 100*;*
COMMIT;

The records of four fictitious clients prior to the update are shown in figure 3. The records of Client 1 and Client 2 are located on page 1 of the database file and those of Client 3 and Client 4 on page 2. Each client's account has not yet been incremented by R50.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	950
	Client 2	68	2100
Page 2	Client 3	77	40000
	Client 4	89	840

Figure 3: The data values before the execution of the update transaction.

The resulting values after the successful execution of the update transaction is shown in figure 4 .

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	1000
	Client 2	68	2150
Page 2	Client 3	77	40050
	Client 4	89	890

Figure 4: The data values after the execution of the transaction.

In general, if a system failure occurs during the execution of the update transaction, the resulting database can be in any of the following states:

1. The writes were logically complete, but the physical writes had not been initiated. The resulting database is in the state depicted by figure 3, with the accounts of all the clients containing the values prior to the transaction.
2. The write was both logically and physically complete, as in figure 4. Here, no loss of information occurs. The accounts of the clients all contain the additional R50 as result of the transaction.
3. The write was in progress when the failure occurred. The data values on page 1 are updated before those on page 2. Assume that the write to page 2 was in progress when the failure occurred, resulting in an inconsistent block of data, with some current and some invalid old data. The amounts shown in figure 5 reflect the accounts of the clients when the failure occurred. The accounts of Client 1, Client 2 and Client 3 are current, but not that of Client 4.

The state of the system no longer reflects a consistent database. It is not possible to reconstruct a previous database state from the current state.. If the transaction is re-executed, the accounts of the first three clients will contain R50 too much. Alternatively, if the effects of the transaction are removed from the database, the account of client 4 will incorrectly be decremented by R50.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	1000
	Client 2	68	2150
Page 2	Client 3	77	40050
	Client 4	89	840

Figure 5: Data values during the update prior to crash.

Since no additional data is recorded, it is not possible to determine in which one of the above three states the current database is. An incorrect re-execution or rollback of a transaction will lead to an inconsistent database state. It follows that a mechanism should be implemented to facilitate the recovery from this type of situation, bearing in mind the uncertainty incurred by the UNIX delayed-write policy.

Possible Solutions

A possible solution would be to bypass the UNIX file system and access the data directly. This method is dangerous because it can destroy the consistency of the file system data. The file system algorithms coordinate disk I/O operations to maintain a consistent view of the disk data structure, including the list of free blocks and inode information. The application may write to blocks that are either considered free by the file system or used by the file system. The list of free blocks maintained by the file system may become incorrect, resulting in inconsistent data values. This solution should therefore be avoided.

The UNIX system does provide a file system maintenance algorithm, which performs consistency checks to repair inode references and free block lists after a system failure. This algorithm aims to regain the consistency of the i-list and the list of free blocks, making use of salvation techniques. However, no provision is made for the recovery of data which was residing in main memory when the failure occurred, as depicted in case 3 of example 1. This data will be lost if the application does not explicitly utilize an algorithm which eliminates this problem.

A third solution is to implement a recovery mechanism to supplement the UNIX kernel. This mechanism utilizes the facilities offered by the system and ensures the consistency of files on disk. Such a mechanism is used in the NRDNIX DDBMS and is described in the next section.

2. The NRDNIX DDBMS

The NRDNIX distributed database management system is implemented as a number of processes, called Managers, on a version of UNIX System V. Interested readers are referred to Rennhackkamp [6] for a detailed description of the system.

Access to a local physical database is attained through the UNIX kernel. The Database Manager of the local DBMS interacts with the UNIX kernel via system calls. The kernel uses the block I/O interface when accessing the physical database files on behalf of the Database Manager. All low level operations on the database files are therefore controlled by the UNIX file system, thus abstracting the DBMS from the physical disk accesses.

Operations on the database are grouped into logical units called transactions. A transaction consists of a number of read and write operations on the database records. The Database Manager of the NRDNIX DDBMS controls user space memory management and physical database access. If access to a record is requested, the Database Manager issues the *read* system call. The read operation is performed, the block is read to cache and the transaction is executed. The operations performed on the database are logically written to the log prior to their actual execution. Upon a *write* system call, the updates to data values are first written to the buffer cache and then to the differential file.

Recovery data are accumulated during normal processing. This data is used by the Recovery Manager when recovering from failures. Various recovery techniques are combined to recover to a consistent database in the event of a system failure.

Logical Log Facility

A *logical write ahead log* is implemented. The decomposed relational algebra statements are recorded in a log file prior to the actual execution thereof. Additional information, including the transaction identifier and timestamp, is also logged. New log information is appended at the end of the log file. No physical representation of the data items are logged, thus reducing storage and I/O overhead. The log information is written to disk using the UNIX *write* system call and is therefore subject to the problem incurred with delayed-writes. To minimize this, the buffer associated with the log is flushed at the end of every transaction. If a failure occurs when logically complete log information had not yet been written to the physical log, this can simply be ignored due to the no-undo policy. A *log garbage collection* facility [4] is implemented to remove redundant data from the log.

Checkpointing is used to reduce the amount of work when recovering from a failure. A checkpoint indicates the currently active transactions as well as transactions committed and aborted since the previous checkpoint. In addition, a flush of the NRDNIX database cache is forced. Checkpoint generation is performed at regular intervals to provide a highly consistent database.

Differential Files

The abstraction of the log information from the physical data, achieved by making use of the logical logging facility, implies that no record of the previous state of a data item is kept. This increases the complexity of undoing uncommitted updates and could result in inconsistent data values.

Checkpoint orientated differential files are implemented to overcome this problem. A file consists of two parts: the main file which is unchanged and the differential file which records all the alterations requested for the main file. Only the results of committed transactions are written to the database and the currently active records are maintained in a differential file. Since only the effects of committed transactions remain in the differential file, and only these values are eventually merged with the physical database, the undo of uncommitted values is eliminated [10].

At each checkpoint, a new differential file is opened. Data records currently in use are moved from the current differential file to the newly opened differential file. The contents of the current differential file only include the results of committed transactions. The new differential file now becomes the current copy and updates are performed to this file.

The Bloom Filtering Algorithm

When access to a record is required, the location (i.e. in the differential or main database file) of the record is obtained by making use of a pre-search Bloom filtering algorithm. When a data record is accessed, the filtering algorithm is used to determine whether the block associated with the record is located in the differential file or the main file. This reduces the amount of additional access time usually associated with the differential file method.

A bit vector of length M is kept in main memory. This vector is initiated to 0. In addition, n different hash functions $X_1..X_n$ are implemented. The identifier of a record r is used as a seed to each of the different hashing functions, thus yielding n hash table locations, valued $x_1..x_n$. When a record is accessed in the main file and copied to the differential file, the values at locations $x_1..x_n$ in the bit vector are set to 1. To determine whether a record is in the differential file, the identifier is hashed and the XOR of the corresponding bits at locations $x_1..x_n$ are determined. If the result of the XOR is equal to 1, the record is probably in the differential file. (It is not certain, since filtering errors may occur.) Otherwise, the record is fetched from the main file. The frequency of filtering errors can be controlled by manipulating the values selected for M and n by using a formula described by Severance and Lohman [7].

The Tagging Mechanism

The last operation prior to the closure of the differential file consists of appending a sequence of bytes to the end of the file. This sequence (the *tag* of the differential file) consists of the unique current timestamp associated with the corresponding checkpoint as well as a predefined sequence of reserved characters, thus uniquely identifying the differential file. The tagging mechanism ensures that all the required physical write operations have been performed when a differential file has been successfully closed. The differential file can then be merged with the physical database using the UNIX *read* and *write* system calls.

The Merging Algorithm

The merging algorithm reads a data from the differential file to the UNIX buffer cache. The data are written to the physical database using the delayed-write policy implemented by the *write* system call. Data are transferred in blocks with a size corresponding to a page.

Since a failure may occur while the merge is in progress, a mechanism to indicate the current differential file block that is being merged is implemented. The merging algorithm proceeds as follows:

1. Before the block of data are read from the differential file, a flag which indicates the logical block number in the differential file, is written to a predetermined location on disk.
2. The differential file block is read to the buffer cache and written to the corresponding database file block.
3. The flag is changed to a predefined reserved block sequence indicating that the merge was successful. The flag will always correspond to this sequence if no merge operation is taking place.

The flag is written to a predetermined location on disk. In the event of a system failure, the recovery manager determines which logical database file block was being updated. This ensures that the state of the database prior to the failure can be determined if the failure occurred during the execution of a merge.

Recovery Processing

Upon recovery from a site failure, a redo, no-undo algorithm is executed. All committed transactions that have possibly not yet been propagated, are redone.The following recovery steps are taken:

1. If a merge operation was taking place when the failure occurred, the differential file is remerged.
    - If the failure occurred while either writing the flag to disk or unflagging, the predetermined location will either contain the reserved sequence or the logical block number. If it contains the reserved sequence, the recovery manager will reinitiate the merging algorithm. If the flag was successfully written to disk prior to the failure, only the merging of the block is redone.
    - If a differential file block was in the process of being merged with the database file, the existence of the flag will indicate this. The block corresponding to the flag address is released and the merging is redone. Since the data values will be safely recorded in the differential file, no information is lost.
2. A previous consistent state, consisting of an old copy of the physical database, is constructed. The corresponding differential file is obtained. All closed but not yet merged files are subsequently merged into the main file.
3. The checkpoint record corresponding to the latest differential file is consulted to obtain a

list of the transactions to be redone. These transactions are re- executed.

4. Global system recovery is performed. One of the advantages of a distributed database versus its centralized counterpart is that sites continue processing in the event of failures of other sites. This implies that the updates on local copies of the data must be obtained and executed by the site. Upon recovery, the site obtains the missed updates from other sites and performs the necessary updates to the database [7].
5. After the successful completion of the recovery algorithm, normal processing is resumed.

The NRDNIX Recovery Manager provides the application with a consistent view of the database at all times. The application of the recovery technique is illustrated on the previous example.

Example 2

Consider again the update transaction of example 1:

BEGIN TRANSACTION;
UPDATE *client*
SET *balance* = *balance* + 50
WHERE *age* > 65 AND
balance > 100;
COMMIT;

The record values of the corresponding clients prior to the update are depicted by figure 6 and the result by figure 7. The records of Client 1 and Client 2 are located on page 1 of the database file and those of Client 3 and Client 4 on page 2. The resulting values are located on pages 1d and 2d in a differential file.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	950
	Client 2	68	2100
Page 2	Client 3	77	40000
	Client 4	89	840

Figure 6: Data values before the execution of the update transaction.

If a system failure occurs during the update, the database may be in any of the following states:

1. The writes were all logically complete, but the physical writes have not been initiated. The accounts of all the clients therefore still contain the old values as in figure 6. Upon recovery, the timestamp of the last successfully closed differential file is consulted. The result

PAGE	CLIENT	AGE	BALANCE
Page 1d	Client 1	67	1000
	Client 2	68	2150
Page 2d	Client 3	77	40050
	Client 4	89	890

Figure 7: Data values after the successful completion of the update transaction.

of the update will not be found in the differential file since it had not yet been written to disk. It will therefore correctly be concluded that the results of the transaction had not yet been propagated. The transaction will be redone by using the information recorded in the local log, resulting in the correct values written to all the accounts.

2. The physical write was in progress when the failure occurred, as shown in figure8. The accounts of the first two clients contain the new values R950 and R2100 and those located on page 2d still contain the old values. These pages are located in the differential file, since only the effects of closed differential files are merged with the main database file. The physical database was not influenced. Upon recovery, the contents of the incomplete differential file are discarded and the update is redone by using the log information.

PAGE	CLIENT	AGE	BALANCE
Page 1d	Client 1	67	1000
	Client 2	68	2150
Page 2d	Client 3	77	40000
	Client 4	89	840

Figure 8: Data values during the execution of the update transaction.

3. The write was both logically and physically complete, but the merge had not yet been executed. Again, the updated records were written to the differential file and the main database was not effected. The differential file thus contains the correct amounts for all the relevant clients shown in figure 7. If the differential file was already closed by using the previously described method, it is merged with the main database file, the result being a consistent and current database. If not, the

database is rolled back to the state prior to the beginning of the differential file and all the relevant transactions, including the update, are redone by using the log.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	1000
	Client 2	68	2150
Page 2	Client 3	77	40050
	Client 4	89	840

Figure 9: Data values during the execution of the merging algorithm in the main database file.

4. If the differential file and the main database file were being merged as shown in figure 8 when the failure occurred, the effects of the update are recorded in the local log. The unmerge algorithm is executed. The database is rolled back to the state prior to the beginning of the differential file and all the relevant transactions, including the updates, are redone by using the log.
5. If the merge was successfully completed, as shown in figure 10, the data values in the accounts of all four clients are up to date. The transaction's effects are valid and secure.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	1000
	Client 2	68	2150
Page 2	Client 3	77	40050
	Client 4	89	890

Figure 10: The resulting data values in the main database file after the merge process.

The example illustrates that the recovery algorithm is sufficient to recover from any of the above situations. The amounts in the clients' accounts are correct and no information is lost when using this scheme.

Conclusion

The loss of information due to the delayed-write policy of the UNIX buffer cache is eliminated by the use of logical logging and differential file recovery techniques. Committed, but unpropagated, transactions are redone by executing the described recovery algorithm. The merging algorithm is used to construct a previous consistent database state by combining the main database file and all the closed, but not yet merged, differential files. The use of a tagging mechanism ensures that only the effects of committed transactions are written to the physical database. The tag also indicates the last complete differential file in the system. The timestamp of this differential file is then used to obtain the corresponding checkpoint record, thereby ensuring that the necessary transactions are redone. In the event of a failure during the merge process, the unmerge algorithm is used to construct a previous consistent state. This ensures the reconstruction of a previous consistent database state.

Differential files can be used when implementing an incremental dumping recovery facility to facilitate the recovery from disk failures. Closed differential files are written to auxilary storage together with a previous consistent database file. In the event of a disk failure, a previous consistent database is constructed by executing the merging algorithm on the files located on the auxilary storage. This increases the reliability of the DDBMS.

The above mechanisms overcome the shortcomings of UNIX and are implemented as user processes. They interface with the UNIX kernel through standard system calls. It was not necessary to modify the UNIX kernel. The DDBMS, developed in C, is therefore highly portable and system independent. One of the main objectives of the NRDNIX DDBMS, namely to provide a consistent, but highly portable system, is therefore met.

References

[1] J A Anyanwu, [1985], A Reliable Stable Storage System for UNIX, *Software - Practice and Experience*, 15(10).

[2] M J Bach, [1986], *The Design of the Unix Operating System,* Prentice-Hall International, Inc.

[3] A Borg et al, [1989], Fault Tolerance under Unix, *ACM Transactions on Computer Systems*, 7(1), 1- 24.

[4] A Deacon, [1988], Global Consistency using Logical Logs, *Technical Report*, Department of Computer Science, University of Stellenbosch.

[5] D M Ritchie, [1979], The UNIX I/O System, *UNIX Programmer's Manual*, Seventh Edition, 522-528.

[6] M H Rennhackkamp, [1990], The NRDNIX Distributed Database Management System, *South African Computer Journal*, 1(1), 5-10.

[7] D G Severance and G M Lohman, [1976], Differential Files: Their Application to the Maintenance of Large Databases, *ACM Tranactions on Database Systems*, 1(3), 256-267.

[8] M Stonebraker, [1981], Operating System Support for Database Management, *Communications of the ACM*, 24(7), 412-417.

[9] M Stonebraker, [1988], Problems in Supporting Transactions in an Operating System Transaction Manager, *ACM Operating System Review*, 19(1), 6-14

[10] H L Viktor, [1988], Distributed Database Recovery, *Proceeding of the RCP Group Conference for Masters and Ph.D. Students in Computer Science.*

[11] P J Weinberger, [1982], Making UNIX Operating Systems Safe for Databases, *The Bell System Technical Journal*, 61(9), 2407-2423.

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as a Communications or Viewpoints. While English is the preferred language of the journal papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted **in triplicate** to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines.

- Use double-space typing on one side only of A4 paper, and provide wide margins.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.
- Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.
- References should be listed at the end of the text **in alphabetic order** of the (first) author's surname, and should be cited in the text in square brackets. References should thus take the following form:

  [1] E Ashcroft and Z Manna, [1972], The translation of 'GOTO' programs to 'WHILE' programs, *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, 250-255.

  [2] C Bohm and G Jacopini, [1966], Flow diagrams, Turing machines and languages with only two formation rules, *Comm. ACM*, **9**, 366-371.

  [3] S Ginsburg, [1966], *Mathematical theory of context free languages,* McGraw Hill, New York.

Manuscripts *accepted* for publication should comply with the above guidelines, and may be provided in one of the following formats:

- in a **typed form** (i.e. suitable for scanning);
- as an **ASCII file** on diskette; or
- as a **WordPerfect, TEX** or **LATEX** or file; or
- in **camera-ready** format.

A page specification is available on request from the editor, for authors wishing to provide camera-ready copies. A styles file is available from the editor for Wordperfect, **TEX** or **LATEX** documents.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

South African Computer Journal

Number 4, March 1991
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 4, Maart 1991
ISSN 1015-7999

Contents

GUEST EDITORIAL

Does Today's Industry Need Qualified Computer Scientists?
Viewpoint I : **H S Steiner** 1
Viewpoint II : **P Visser** 2

RESEARCH ARTICLES

Hypertext for Browsing in Computer Aided Learning
J Barrow 4

CID3: An Extension of ID3 for Attributes with Ordered Domains
I Cloete and H Theron 10

The Universal Relation as a Database Interface
M J Philips and S Berman 17

Database Consistency under UNIX
H L Viktor and M H Rennhackkamp 25

An Interrupt Driven Paradigm of Concurrent Programming
P Clayton 34

An ADA Compatible Specification Language
R Bosua and A L du Plessis 46

Knowledge-Based Selection and Combination of Forecasting Methods
G R Finnie 55

A Causal Analysis of Job Turnover among System Analysts
D C Smith, A L Hanson and N C Oosthuizen 64

An Analysis of the Usage of Systems Development Methods in South Africa
S Erlank, D Pelteret and M Meskin 68

COMMUNICATIONS AND REPORTS

Book Reviews 78
Editorial Comment 80
Automatic Vectorisation
L D Tidwell and S R Schach 81