Databases: Improving Usability and Responsiveness

EDITED BY
BEN SHNEIDERMAN

DATABASES: IMPROVING USABILITY AND RESPONSIVENESS

BEN SHNEIDERMAN

Department of Information Systems Management
University of Maryland

ACADEMIC PRESS New York San Francisco London 1978

A Subsidiary of Harcourt Brace Jovanovich, Publishers

DATABASE CONSISTENCY AND INTEGRITY IN A MULTI-USER ENVIRONMENT

Michael F. Challis[1]

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro
Brasil

1. INTRODUCTION

In this paper we propose a solution to the twin problems of data base consistency and integrity that is based on a particular "storage" model: briefly, an extra block-to-block mapping is inserted between the "logical" data base and the physical data file that contains it, thus providing a simple means of economically representing several similar "instances" of one data base at the same time. The basic technique has been used by the author to provide integrity in a single-user environment for the JACKDAW data base package ([2], [3]) and has also recently been reported independently by Lorie ([9]).

Section 2 introduces the problems of consistency and integrity, and briefly discusses conventional solutions such as "logging" and "roll-back" features. Section 3 describes the model on which the technique is based, and Section 4 explains its use in a single-user environment. We show how a user may design his applications programs to ensure whatever degree of consistency he requires, and how he may test new applications programs safely on a "live" data base without compromising its integrity.

Sections 5 and 6 extend the technique for use in a multi-user environment. In Section 5, the concept of "indivisibility"

[1]Partially supported by the Brazilian government agencies FINEP and CNPq.

ISBN 0-12-642150-1

is carefully examined, and a user primitive *indivis(t)* is suggested which allows a process *P* to attempt to execute a transaction *t* without interference from other concurrent processes, and in such a way that any effects of *t* appear to occur at a single moment from the point of view of those other processes. In the case where conflict between processes is unlikely, such an attempt will normally be successful, and we believe it to be a useful alternative to, say, the use of semaphores or record "locks". Other facilities for testing sets of co-operating processes on live data bases, and for accessing "snap-shots" of data bases from which consistent reports can be taken are also suggested. Section 6 develops an implementation of these features to show that the cost involved is not high provided that the various instances concurrently represented are not too divergent.

2. CONSISTENCY AND INTEGRITY

In this section, we define the properties of consistency and integrity as applied to data base systems; for a more detailed discussion of these concepts, references [10] (see sections 2.5 and 2.6) and [5] (chapter 24) should be consulted.

A data base is said to be *consistent* at a particular moment if all the semantic constraints governing allowable relationships between records, values of fields, etc. are satisfied. Some examples may help to illustrate this concept:

a) A new record with a particular key is to be added to a data base. The data base is not consistent until both the new entry has been added, and the index for the key field has been updated.

b) A record is to be deleted. The data base is not consistent until all references to that record have also been removed (such references include entries in indexes as well as pointers from other records).

c) A travel agent is booking a holiday. The data base is not consistent until the outward flight, hotel and return flight have all been reserved.

The simplest way to ensure consistency is to arrange that any sequence of interdependent updates is *indivisible*: that is, no other process (including a read-only process) may have access to the data base until the sequence is completed.

In a single-user environment this is easy to achieve, provided we have perfect hardware and software; for then each run of an applications program is indivisible, and so we need only make sure that each program leaves the data base consistent before terminating.

But hardware and software are never perfect, and so there is always the possibility that any program may be interrupted at an arbitrary moment, thus leaving the data base in an arbitrary and possibly inconsistent state. (For example, the program may simply run out of time.) Considerations such as these lead to the concept of integrity: a data base system is said to have *integrity* if, after an arbitrary halt, the data base may be recovered to some consistent version.

One common way to provide data base integrity is by means of logging, check-point and roll-back facilities (see [16] for example). Every change to the data base is recorded on a separate logging file, and, at appropriate moments, each application program writes a "check-point" record to the log to note that the data base is consistent. When restarting after a failure, the system uses the information recorded on the log to "roll-back" the data base to a consistent state. This paper describes a different solution to the problem which does not require logging and roll-back facilities; further advantages of this solution are the provision of extra facilities, such as the ability to test new programs safely on a live data base.

3. THE BASIC DATA BASE MODEL

Before proceeding to details of the solution adopted, we describe the "physical" (or "storage") model of the data base upon which the solution depends.

We suppose that the data base is composed of a sequence of data blocks, all of the same size, which are referenced by

sequence number. Any physical pointers within the data base (between records for example) are represented by "block number/ offset" pairs, and so the unit of access is the block which is referenced by sequence number. This is, of course, a representation of the data base at a very low level. The interface to the user will be in terms of records and fields, and is unlikely to include any references to physical locations. It should also be understood that the correspondence between physical blocks and logical records is not necessarily one-to-one.

In order to develop a solution to the integrity problem we insert an extra mapping between this model of the data base and the "data file" itself on disc. The data file, too, is composed of a sequence of fixed size blocks, but each block of the data base is not necessarily associated with the corresponding block in the data file. For the remainder of this paper we will use the word *logical* to refer to aspects of the data base model and *physical* to refer to aspects of the data file; thus the data base is modelled as a sequence of *logical blocks* which are mapped onto the *physical blocks* of the data file. The one-to-one mapping which defines this correspondence is called the *LP mapping*.

e.g. A logical data base of four blocks might be represented in a physical data file of six blocks as follows:

0	1	2	3	4	5
L_0	L_2		L_3	L_1	

The LP mapping is:

LP(0) = 0; LP(1) = 4; LP(2) = 1; LP(3) = 3

We say that physical blocks 2 and 5 are *spare*.

We see that a logical data base is defined by a physical data file together with an LP mapping. In particular, one data file may represent more than one data base, and it is this possibility that provides the key to the integrity technique presented in this paper.

The technique may be extended in an obvious manner to the case where a logical data base is represented on several data files with different block sizes; this covers several well-known commercial data base systems such as ADABAS ([14]) and TOTAL ([15]).

4. THE SINGLE-USER ENVIRONMENT

4.1 The Integrity Technique

This section briefly describes the technique used to ensure integrity in a single-user environment. As mentioned in Section 1, this technique was independently proposed by Lorie, and the reader is referred to [9] for a more complete presentation.

For the sake of argument, we suppose that the LP mapping defining the logical data base is held in a physical block specially reserved for this purpose, called the *root block*. Then the data file is self-defining in the sense that a particular instance of the logical data base is defined by the contents of the root block; we call this the *disc instance*.

Each indivisible section of a program is executed in the following steps:

i) Initialisation. The LP mapping describing the disc instance is copied into core; we call the data base defined by the core LP mapping the *current instance*.

ii) Updating. Whenever a logical block is updated for the first time within this section, a new spare physical block is allocated to it, and the LP mapping in core is appropriately updated. A note of the number of the physical block previously allocated is added to a list of "pending" blocks. (Such pending blocks belong to the disc instance but not to the current instance.)

iii) Flushing. At the end of the section, the modified LP mapping in core is written back to the root block, so that the current instance becomes the new disc instance. Blocks in the pending list are now no longer needed and are marked as spare.

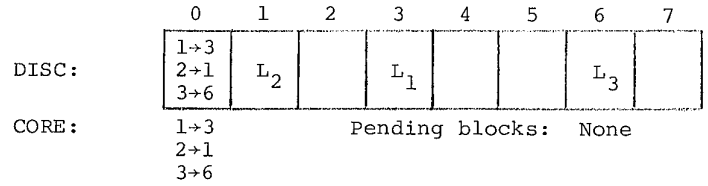
The cycle is repeated to execute the next indivisible section, although it is not, of course, necessary to repeat step (i).

The effect is that the disc instance changes directly from one consistent state to the next; any intermediate steps are defined only by current instances in core, which are lost when a program terminates. So integrity is assured: if a program

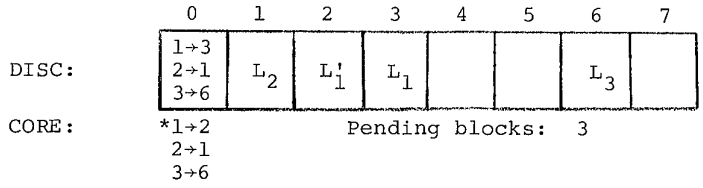
terminates unexpectedly within an indivisible section, then the instance made available when the data file is next opened will be that corresponding to the consistent state in force just prior to entering the interrupted indivisible section.

As an example, consider the following situation where the indivisible section includes two updates to logical block 1 and one to logical block 3:

The initial state is as follows (block 0 is the root block):

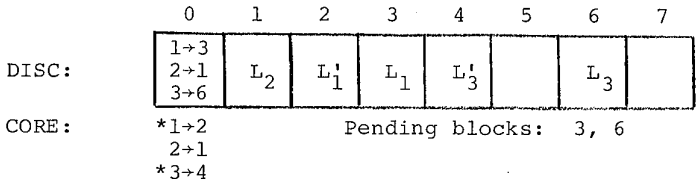


To update block 1, we first read physical block 3 into core and alter it; since this is the first change to this logical block within the section, we assign a new physical block before writing the altered logical block back to disc - in this example we choose physical block 2. We record the previously allocated block (number 3) in the pending list:

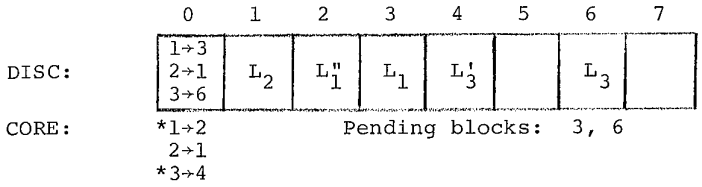


(* An asterisk against a mapping element indicates that the logical block has been altered during the current section.)

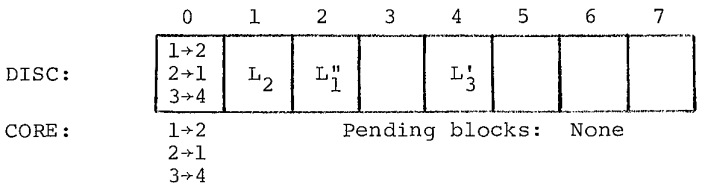
Next, we update logical block 3 in the same way:



We now make a further alteration to logical block 1. This is not the first change, and so there is no need to assign a new physical block:



The section is now complete and we rewrite the modified mapping to the root block, and mark the pending blocks 3 and 6 as spare:



Note that this technique is efficient in the sense that we do not need to make extra copies (or "before-images") of logical blocks before altering them, since the original versions continue to exist until they are no longer needed. In a similar way, the recovery process is non-existent, since there is no inconsistent version of the data base to roll back.

On the debit side, the physical data file must include enough spare blocks to record new versions of all blocks updated during a single section. In the worst case, this would require the file to be twice the size of the logical data base, but in practice only a small portion of a large data base is likely to be altered indivisibly.

A different solution to the integrity problem is provided by the "differential file" concept ([13]), where records modified in a large data base are stored separately in a *changes* file instead of overwriting their originals in the *master* file. Our technique can be viewed as an extension of this in which the master and changes files are merged together.

4.2 Practical Considerations

The implementation illustrated above has been deliberately simplified for didactic purposes, and there are several improvements which may be made in practice.

a) Core buffers. Normally, there will be several core buffers available to hold copies of data blocks. Any core buffer whose contents differ from the corresponding disc block must be written back whenever a new disc instance is recorded.

b) Alternate root blocks. It is better to allocate two physical blocks which are used alternately as root blocks. In this way, the mapping defining the previous disc instance is not overwritten when the new LP mapping is written to disc at the end of a section; so if an I/O error occurs during this transfer, the previous disc instance can still be recovered.

c) Large data bases. In a data base of any size, it is unlikely that the LP mapping will fit into a single block, but it is a simple matter to extend the scheme, permanently allocating two separate sets of physical blocks to be used in turn for holding the disc instance mapping.

With large mappings it also becomes increasingly inefficient to hold the entire mapping in core and to write it back when flushing at the end of each indivisible section. The solution to this is to "page" the mapping itself, being careful to allocate new physical blocks to altered logical blocks in much the same way as new blocks are allocated to updated data blocks in the simple scheme.

4.3 User Facilities

4.3.1 Flushing and Indivisibility. We have so far assumed that *flushing* (that is, defining a new disc instance) takes place whenever (and only when) an indivisible section is completed. We believe that it is better in practice to separate these concepts, providing the following facilities for the user:

flush() - this procedure call causes a new disc instance to be recorded.

indivis(p) - this prevents any further flushing until procedure *p* has been executed.

Unless a call of *indivis* is active, the package is free to flush whenever it likes: for example, the system might choose to construct a new disc instance every 30 seconds, or after every 200 updates, or whenever the number of spare blocks

available is less than 50. In particular, the system would probably choose to flush immediately before entering an indivisible section in order to make available as many spare blocks as possible, and will certainly flush before closing the data file upon the successful completion of an update run. We also assume that each basic data base access procedure is indivisible.

The user now has available a choice of techniques for achieving the degree of consistency that he desires.

A simple program which applies a sequence of independent updates may choose not to use either *flush* or *indivis*; it may safely be re-run after an error, since a second application of the same update can do no harm. A more sophisticated version of the same program might make regular calls to *flush*, recording on its own "log" data set the number of the last input record processed prior to the flush, thus reducing wasted processing time if it needs to be re-run. This log data set may, of course, be the data base itself, and this suggests the provision of extra procedures which make it easy for a program to record messages about its own progress:

recordmessage(identifier, message)
- to associate *message* with *identifier* in the data base.

message := readmessage(identifier)
- to read the *message* associated with *identifier*; a special value is returned if no such message exists.

message := readdelmessage(identifier)
- to read and then delete any message associated with *identifier*.

By associating a common message identifier with a suite of programs it is easy to ensure that all are executed in a defined sequence: upon successful completion, each program assigns a new value to the identifier, which will be checked by the next program in the sequence.

The *indivis* procedure would be used by a more complex program which applies a sequence of "transaction" updates. Each transaction is composed of a number of dependent updates, but is independent of other transactions. Flushing may or may not be explicitly requested as in the previous example.

Finally, a very time-consuming series of dependent updates

can be split into smaller sections, each of which is executed indivisibly. At the end of each section, progress is noted using *recordmessage*, and the data base is then flushed. If this approach is adopted, other programs must be aware of the corresponding message identifier and should check its value before proceeding, in case the update program had failed to complete successfully when it was last run.

4.3.2 Testing programs. If we treat the whole of a program as an indivisible section, and do not flush the data base before terminating, then the data base defined by the disc instance will remain unaltered. In other words, we are able to test update programs on "live" data without compromising its integrity. A similar facility is suggested in [11] in the context of differential files. This special case is so important that a "test only" mode of opening a data base should be provided.

5. THE MULTI-USER ENVIRONMENT

5.1 Indivisibility

In this section we examine the meaning of *indivisibility* in a multi-user environment in some more detail, and suggest a multi-user analogue of the *indivis* procedure defined for the single-user case.

Suppose that we have a set of concurrent processes accessing a data base, and that one of them (say P) specifies a transaction t that is to be executed indivisibly. One way to do this is to halt all other processes allowing only P to continue, but this may be needlessly inefficient; for example, if the purpose of t is to reserve a seat on an aeroplane flight, then only those processes accessing the same aeroplane seat actually need to be halted.

As t is executed, references will be made to fields and records in the data base and decisions will be taken based on the values found; t will then usually record these decisions in the form of alterations to the data base.

We can think of the information upon which t bases its decisions as its *requirement*, and the alterations that it makes may be called its *effect*. We can represent a requirement as the union of a sequence of conditions on the records, fields, relationships etc. that are represented in the data base:

e.g. (record X exists) & (field Y of record X = 25) &
(record A points to record B) & ...

and an effect may be represented as a sequence of updates:

e.g. (create record Y) & (field Z of record Y := 24) & ...

We may now state more precisely those conditions under which a process P' may be allowed to execute concurrently with P whilst t is active:

i) P' must not alter the validity of t's requirement, for such alterations may invalidate t's decisions.

ii) P' must not access records or fields that take part in t's effect; if it does, it may itself take erroneous decisions based on an inconsistent view of the data base.

Note that a transaction's requirement may require that a particular record does not exist, and that in this case condition (i) above means that P' must not create that record. Such records are called *phantom* records, and the concept of *predicate locking* has been described to handle such cases (see [7]). A predicate lock for a transaction t essentially defines a logical area of the data base to which other processes must be denied access (or given restricted access) whilst t is active. The notions of requirement and effect may be viewed in the same way: the effect defines the logical area which must be denied completely to other processes, and the requirement defines a "read-only" area.

One way to ensure that conditions (i) and (ii) above are complied with is to use the *critical section* concept (see [6]). Those sections of each process that might violate either condition are designated as mutually exclusive critical sections, and semaphores are used to guarantee that at most one process is executing within a critical section at any particular moment. The difficulty here lies in attributing suitable semantics to each semaphore that is to be used, and in ensuring that all processes (including those to be written in the

future) obey the rules that have been chosen.

A simple allocation of semaphores will often result in unnecessary sequencing: for example, if the seat reservation transaction is guarded by a semaphore *S*, then two instances of this transaction will never execute concurrently even if they are for different flights. This suggests the provision of certain "standard" semaphores by the data base system itself, such as one or more "access" semaphores associated with each record. Commonly two are provided: the system uses one to sequence all accesses to the record, and the other to sequence update accesses only. These semaphores may be accessed by a user process through *lock* and *unlock* primitives, which allow a process *P* to gain *exclusive* access to a record *R* (no other processes may access *R*) or *shared* access (other processes may read *R*).

As with semaphores, the general use of locks introduces the possibility of "deadlock", where two or more processes become mutually blocked, each one attempting to lock some part of the data base that has already been locked by one of the others (see [4] for a general discussion). When this occurs, one or more of the processes must be "backed-out" in order to allow others to continue.

The problem of locking a phantom record may be solved (albeit clumsily) by exclusively locking all records of the appropriate type, and so a common generalisation of record locking is to provide locking facilities of a coarser "granularity" controlling access to certain sets of records such as:

- all records referenced by record *R*
- all records of type *T*
- all records in the data base

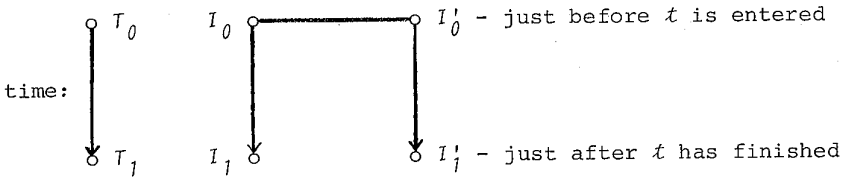
In [8], a protocol based on such a hierarchy of lockable objects is presented. Any process wishing, for example, to lock a record within this protocol must first place an "intention" lock on all higher level objects in the hierarchy. This makes it easy for the system to determine whether a particular request is compatible with other requests already granted, and simplifies the detection and resolution of deadlocks.

Here we suggest a different technique which requires only that transactions are marked as indivisible by use of the

indivis procedure:

b := *indivis*(t)

At time T_0 when this call is made, a copy I'_0 is made of the current instance I_0 of the data base. The transaction t is then applied to the copy I'_0 to produce a new instance I'_1 at time T_1; in the meantime, other processes operating on the current instance I_0 have produced a possibly modified instance I_1:



During execution of t we keep a note $R(t)$ of those records and fields (and their values) read by t from the initial instance I'_0: it is clear that $R(t)$ will include t's requirement. We also keep a note of any alterations made by t in $E(t)$, which thereby represents t's effect.

At time T_1 we examine I_1 to see if items mentioned in $R(t)$ and $E(t)$ have remained unaltered since time T_0. If so, we may safely apply the alterations $E(t)$ to I_1, the effect being as if t had, indeed, been executed indivisibly at time T_1; we say that the call of *indivis* has been successful, and return a result of *true*. If not, the call is unsuccessful, and a result of *false* is returned. (A special case arises if $E(t)$ is empty; in this case, t has no effect and so its execution is "invisible" to the other processes. So we may suppose that it was indivisibly executed at time T_0 and return a result of *true*.)

In the proposed realisation of *indivis* (see section 6.2) the data base is copied by copying the LP mapping, and the sets $R(t)$ and $E(t)$ are represented as lists of those logical blocks referenced and updated by t. A new physical block is allocated whenever a logical block is updated, and so it is easy to determine whether $R(t)$ and $E(t)$ remain unaltered in I_1, and, if so, to apply $E(t)$ to I_1: we simply alter the LP mapping for I_1 to reflect the changes made. Note that phantoms are accounted for, since the block in which a phantom might appear will be recorded in $R(t)$ when t examines the data base to see if the record exists. These considerations show that it is indeed possible to implement *indivis* in an efficient manner.

Choosing the block as the unit of representation for requirement and effect means that certain transactions which could logically execute concurrently will be prevented from so doing. (For example, two transactions which update separate records which happen to lie in the same block.) The importance of this will clearly depend on the size of the block and the distribution of transactions, but a recent paper ([12]) suggests that a large "granule" size is often more efficient than a small one when the locking overhead (as in this case) is less.

How does *indivis* compare with a more conventional approach using locks? We saw above that transactions competing for locks may enter deadlocks which can only be resolved by backing out one or more of the processes; such a situation may be further complicated by the fact that another transaction may already have taken a decision based on values recorded by the transactions to be backed out, and so should itself also be backed out. To avoid this "snowball" effect it is usual to insist that a transaction exclusively locks those records that it updates, only releasing them upon termination. Such precautions are not necessary with *indivis*, since the essence of the technique is the concurrent existence of separate instances of the data base. If we suppose for the moment that all processing is by means of indivisible transactions, then a "deadlock" corresponds to the existence of "incompatible" instances; it is "resolved" when the first transaction to complete incorporates its corresponding instance into the current instance, and the other processes are "backed out" when they complete unsuccessfully and their corresponding instances are abandoned.

Another advantage of *indivis* is that the programmer is no longer responsible for specifying the area of the data base to which a transaction is "sensitive"; this area is instead determined dynamically by the system in the sets $R(t)$ and $E(t)$. So the programmer is protected from mistaken assumptions about process interactions, and is less likely to corrupt the data base; on the other hand, undisciplined use of *indivis* may result in much wasted processing time if many calls are made before a transaction succeeds.

It is clear that *indivis* is most appropriate in circumstances where conflict is unlikely. A suitable candidate might

be the seat reservation system where many copies of the reservation process are executing concurrently. Each reservation is recorded indivisibly, and will only fail if another concurrent reservation for the same seat completes first.

In a system where conflicts are common, explicit sequencing (using semaphores or locks) is indicated. For example, suppose a data base contains details of tickets for a show which are to be allocated sequentially. If several processes are concurrently processing ticket applications, it is clear that the critical sections in which a ticket is allocated are always mutually exclusive, and it is never sensible to attempt to execute two such sections concurrently. In this case, the constraint is simple and the system designer may use a single semaphore to force sequential ticket allocation.

It is interesting to relate *indivis* to the facilities offered by System R ([1]), where a user may associate a particular "level of consistency" with each transaction t as follows:

level 3 - t is indivisible.

level 2 - changes made by a concurrent transaction t' are only made available to t when t' terminates.

level 1 - t sees changes made by concurrent transactions as they happen.

An additional constraint applied to all levels is that any data altered by one transaction will not be altered by any other until the first has completed. (This makes it possible to "back out" one transaction when a deadlock occurs without undoing the effects of any other.)

indivis corresponds to a level 3 transaction, except that it might fail. (In System R, locks are applied to enforce the various consistency levels, and all transactions will eventually complete: transactions backed-out from deadlocks are repeated as necessary.) Processes accessing the current instance are more like level 2 transactions in that they only see changes made by successful *indivis* transactions, but (unlike System R transactions) they may (and will) freely interact with each other. There is no equivalent to the level 1 transaction (which in System R may even see changes that are later "undone").

5.2 Integrity

As in the single-user environment, all basic calls to the data base system are indivisible, and the procedures *flush*, *recordmessage*, *readmessage* and *readdelmessage* are provided with identical definitions. (Note that calls of *recordmessage* and *readdelmessage* may be used as *V* and *P* operations on a binary semaphore because they are indivisible.)

The *indivis* procedure defined in the last section may also be used simply as a consistency aid: if a process terminates unexpectedly in the middle of an indivisible section, then we can be sure that no part of the effect of that section has been incorporated into the current instance of the data base, and hence cannot possibly appear in the disc instance.

As before, individual processes may make calls of *flush* to guarantee that the disc instance will at least reflect progress up to a certain point, and/or may record progress using *recordmessage*.

5.3 Testing Programs

One possibility is to define a procedure *test(p)* which applies *p* to a copy of the current instance; when *p* terminates, the copy is thrown away. (*test* is similar to *indivis* except that no attempt is made to incorporate the modified copy into the current instance upon completion.) This however, allows us to test only single processes, and so we suggest a more powerful facility which allows the user to create and access *secondary* versions of the data base. Each such secondary version starts life as a copy of the current instance of the *primary* version, and is then modified independently by the set of processes under test. Secondary versions differ from the primary version in that there are no disc instances associated with them: in other words, a secondary version is lost when the last process accessing it terminates.

5.4 Read-only Access

A common problem in a multi-user environment is that of obtaining consistent reports. For example, consider the case of a program which generates a summary of items in stock followed by a detail report showing the location of these items by warehouse. If stock figures are updated whilst this program is being executed, the totals in the two reports will not tally.

This problem may be solved by specifying the entire program as an indivisible transaction; in this way, the program will operate on an (unchanging) copy of the current instance. Since the transaction has no effect (in the sense of section 5.1) the call of *indivis* will always be successful. This is such a common requirement that we suggest a special mode of "read-only" access in the next section.

5.5 User Facilities

This section gives a formal description of the facilities suggested above.

When a process first requests access to the data base, it must specify both the *version* and the *access mode* desired:

open(version, mode)

version = 0 means that the process is a production program which is to access the primary (i.e. "live") version of the data base. This is the only version represented by the disc instance.

version = *n* (>0) means that the process is to be tested on a secondary version of the data base. If version *n* does not yet exist, it is created by taking a copy of the current instance of the primary version.

mode = 0 means that the process wishes to access the current instance of the specified version at all times. It is allowed to alter this instance.

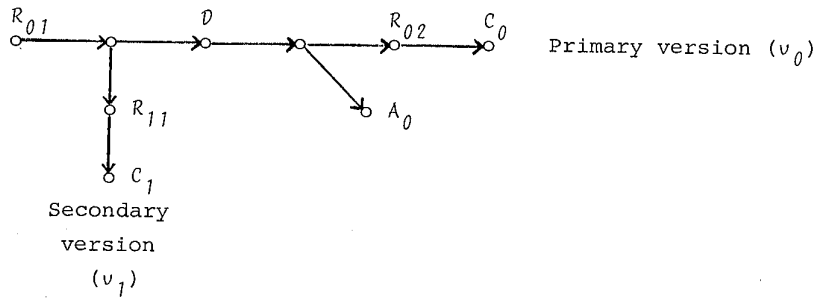
mode ≠ 0 means that the process is read-only, and is to be applied to a copy of the current instance of the specified version.

The procedures for manipulating instances are:

flush() - this has no effect unless it is applied to the current instance of the primary version. In this case, a new disc instance is recorded.

b := *indivis*(*p*) - if this is applied to the current instance of a version, the procedure *p* is applied to a copy of this instance and an attempt is then made to incorporate the result into the current instance once again. The result is *true* if and only if the attempt is successful. If it is applied to any other (read-only) instance, the effect is simply to execute *p* and return *true*.

We can represent the relationships between the various instances of a data base that exist at a particular moment as follows:



In this example there is one secondary version with two active instances and five active instances of the primary version. R_{01}, R_{02} and R_{11} are read-only versions, and C_0 and C_1 are the current instances of the two versions. D represents the most recent disc instance (created by a call of *flush*) and A_0 represents a currently active *alternative* instance created by a call of *indivis* that has not yet completed.

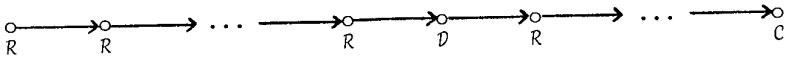
6. IMPLEMENTATION

This section outlines a possible implementation stage by stage. We first permit access to the primary version only, showing how read-only instances and flushing can be implemented

next we show how to manage alternative instances, thus realising *indivis*. Finally we indicate how to manage secondary versions.

6.1 Read-only access and Flushing

6.1.1 Introduction. At any moment, the various active instances of the data base can be represented in order of creation as:



where each R is a read-only instance, D corresponds to the disc instance, and C is the current instance.

New read-only instances are created by calls of *open*, and an existing read-only instance is released when its associated process terminates. The current instance is created when the first process opens the data base, and is released when the last process terminates. The disc instance is always present: when the data base is flushed, a new disc instance is created and the previous one is released.

Associated with the data base is a positive integer known as the current *epoch* which is increased by one whenever a new instance is created. The value of this integer at the time of creation of an instance is known as the *epoch* of that instance, and may be used to identify it. Note that the epochs of the active instances of a data base are not necessarily consecutive since instances are not necessarily released in the same order as they were created; but $epoch(I)$ is always less than $epoch(J)$ for an instance I older than J.

Each instance I_k of epoch k is defined by its LP mapping LP_k. A new current instance I_{n+1} is created by creating a new mapping LP_{n+1} equal to LP_n. As soon as logical block ℓ of I_{n+1} is altered for the first time, a new physical block p' is assigned to it in LP_{n+1} so that the alteration appears only in instance I_{n+1} and not in any preceding instances. The physical block p previously assigned to ℓ cannot be marked as spare, since it is still a part of I_n, and, possibly, of other preceding instances; indeed, it can only be reused when all

instances of which it is a part have been released.

In general, a physical block p is in one of the following states:

a) in *current* use, if $LP_n(\ell) = p$ for some ℓ, where n is the epoch of the current instance.

b) *pending*, if it is not in current use, but there exists an active instance with epoch k such that $LP_k(\ell) = p$ for some ℓ.

c) *spare*, if it is neither current nor pending.

When a logical block of the current instance is updated for the first time, the newly assigned physical block changes state from spare to current, and the original block from current to pending. Each pending block belongs to a sequence of one or more active instances, and becomes spare when all the instances of that sequence have been released.

We keep track of pending blocks by recording their epoch of allocation $alloc$ and epoch of release rel: a pending block p becomes spare as soon as there is no active instance I such that:

$$alloc(p) \leq epoch(I) < rel(p)$$

$alloc(p)$ is recorded in the physical block itself at the time of allocation. At the time of release, this value is used to determine the first instance I to which p belongs; this instance is given by:

$$epoch(H) < alloc(p) \leq epoch(I)$$

where H is the active instance immediately prior to I. The pair $(p, rel(p))$ is then added to a list associated with I.

In this way, a list is kept for each instance I of those pending blocks which belong to I but not to any older instance. When I is released, this list is scanned to see if any pending blocks mentioned can now be made spare; this will be possible if:

$$rel(p) \leq epoch(J)$$

where J is the active instance immediately following I. Pairs $(p, rel(p))$ which do not satisfy this condition are simply added to the list associated with J.

Using this technique, it is easy to keep an up-to-date list of spare blocks available for allocation - that is, of blocks which do not form part of any active instance. But when the

data base is flushed, we must include on disc a list of those blocks which would be spare if no other instances were active: for only the disc instance survives if processing terminates unexpectedly, and we will need to know which blocks are spare when we re-open the data base. This list consists of the union of the set of spare blocks and the set of pending blocks at the time when the new disc instance is made.

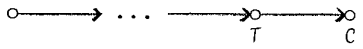
6.1.2 Practical Considerations. The mappings LP_k and LP_{k-1} are likely to share many elements in common, and so a single compact representation for all LP_i can be chosen which allows us to take advantage of this. For example, when the size of the logical data base is large, it becomes necessary to page the LP mapping itself, using a further "LP to physical" mapping (LPP) as an index; in this way each LP_k is defined by LPP_k. This suggests that we keep track of LP instances using the same techniques as those described above for data base instances. For example, copying an LP mapping is reduced to making a copy of the (much smaller) LPP mapping, provided that we allocate a new physical block and alter the LPP mapping describing the current LP mapping whenever one of its blocks is updated for the first time.

Another point concerns the list of spare blocks from which physical blocks are allocated during processing. There is no need to either read this in its entirety when the data base is opened, or to write it all back each time the data base is flushed. It is sufficient to maintain two "windows" on the "front" and "back" of the list: new blocks are allocated from the front window and blocks made spare when an instance is released are added to the back window. The front window is replenished from disc whenever more blocks are needed and the back window is emptied to disc when it becomes full, or whenever the data base is flushed.

6.1.3 Dumping. Even the disc instance will not allow us to recover the data base if it is physically damaged, and so it is prudent to take periodic "back-up" copies on tape. Our technique favours an incremental approach, whereby only those blocks altered since the last dump are copied to tape. These may be determined by comparing $alloc(p)$ with the epoch of the last dump for each block p in the data base: only those created

after the previous dump need to be copied. But such a complete scan of the data base is likely to be unacceptable, particularly if dumping is relatively frequent, and a much better technique is suggested in [9], where an extra bit is associated with each element of the LP mapping to say whether that logical block has been altered since the previous dump. This bit is called the "cumulative shadow bit" in [9]; here we shall call it the *dump* bit, and it is set whenever a logical block is updated.

Whenever the incremental dump process is scheduled, a new read-only "tape instance" T is created:

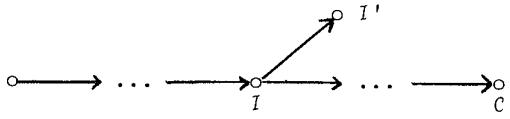


The new current instance C is represented by the mapping LP_C which is a copy of LP_T in which all the dump bits have been cleared.

The incremental dumper may now copy at leisure those physical blocks of T indicated by the dump bits in LP_T; when it has finished, the instance T is released, and any pending blocks required only by the dumping process are automatically made spare. (Thus there is no need for the "long term shadow bit" of [9], whose purpose is to indicate that the corresponding block has not yet been dumped and so cannot be freed.)

6.2 Alternative Instances and *indivis*

We denote the alternative instance that is created from instance I by a call of *indivis* by I':



Associated with I' we keep:

LP' - its LP mapping.

R - the set of logical blocks which includes the information involved in the requirement of I'.

E - the set of logical blocks which includes its effect.

As I' is processed, each logical block that is referenced is added to R, and each logical block that is updated is added to E. New physical blocks are assigned to updated logical blocks in the usual way, so that $LP'(E)$ defines the set of physical blocks that are *local* to I': they belong only to I' and are not shared with any other instance. When the transaction is complete, we must determine whether we can safely incorporate the effect of I' into the current instance C. This will be possible provided that other processes operating on the "main-line" instances have not altered any of the blocks involved in the transaction's requirement or effect: in other words, if the contents of the logical blocks described by $R \cup E$ are the same in instance C as in instance I. This can be easily checked by comparing the appropriate elements of the LP mappings defining the two instances.

If incorporation is possible, the LP mapping for C is updated by reference to E to include the local blocks of I'; if not, the local blocks are returned directly to the spare block list.

6.3 Secondary Versions

The treatment of secondary versions is very similar to that of the primary version; the main difference lies in the treatment of spare blocks.

Each secondary version is essentially treated as a separate entity, with its own epochs unrelated to those of the primary version. As logical blocks of a secondary version are updated, new physical blocks are assigned which are *local* to that version: they can never be shared with any other secondary version or with the primary version. Thus a secondary instance will be composed partly of local blocks and partly of blocks acquired from (and probably shared with) the primary version.

The actions taken when a physical block is released depend on its state. If it is local, it is recorded as a pending block in a list associated with some secondary instance in the usual way. If it is not, no action is taken since we presume that it is still required by the primary version.

7. CONCLUSIONS

This paper has described a solution to the problems of consistency and integrity in large data bases, and a possible implementation has been presented. By giving some details of this implementation we hope we have shown that the technique is efficient provided that the various data base instances that co-exist do not differ drastically one from the other.

The technique is based on the provision of an extra block-to-block mapping between a logical data base and the physical data file upon which it resides. In this way, many similar instances of one data base may be economically represented in the same data file by different mappings. But only one mapping (and hence only one data base) is defined in the data file itself: this is the so-called disc instance, which is the only instance preserved when data base processing terminates (whether normally or abnormally). By ensuring that new disc instances are only created when certain consistency constraints are satisfied, we can ensure the integrity of a data base across unexpected system and application program failures.

An extension of the technique is particularly useful in a multi-user environment, and in this paper we have suggested three facilities:

i) The provision of a "frozen" copy of an ever-changing data base (for the use of a report generator, for example).

ii) The provision of a copy of a live data base on which a new program or set of co-operating programs can be safely tested.

iii) The ability to "split" a data base into separate instances: a "main-line" instance and one or more "alternative" instances. The main-line instance continues to be accessed by (possibly several) current processes whereas access to each alternative instance is restricted to the single process *P* that created it.

P is free to make consistent changes to the alternative instance based on decisions about its contents which cannot be affected by the actions of other concurrent

processes. When *P* completes its "critical" task, an attempt is made to combine the two instances: if this is possible without compromising the integrity of *P*'s decisions and alterations, then it is done; otherwise the alternative instance is abandoned, and *P* must try again.

The technique was originally developed solely as a means of ensuring data base integrity in a single-user environment, and is used for this purpose in the JACKDAW data base package. This system has been in use at the University of Cambridge since 1973, supporting an administrative data base containing details of Computing Service users and their resource allocations. During this period, the data base survived unscathed all operating system and application program failures, thus demonstrating the value of the integrity feature. Further development of JACKDAW is now in progress at Pontifícia Universidade Católica in Rio de Janeiro.

ACKNOWLEDGMENTS

The JACKDAW package was designed and implemented whilst the author was employed by the Computing Service at the University of Cambridge, England. Further work has been financially supported by the Brazilian government agencies Financiadora de Estudos e Projetos (FINEP) and Conselho Nacional do Desenvolvimento Científico e Tecnológico (CNPq).

REFERENCES

[1] Astrahan, M.M, et al. "System R: Relational approach to data base management" ACM Trans. Database Syst. 1, 2 (June 1976).

[2] Challis, M.F. "The JACKDAW database package" Proc. SEAS Spring Technical Meeting, St. Andrews, Scotland, April 1974.

[3] Challis, M.F. "Integrity techniques in the JACKDAW database package" Monografia em Ciência da Computação 9/77, Dept? de Informática, Pontifícia Universidade Católica, Rio de Janeiro, Brasil (1977).

[4] Coffman, E.G., Elphick, M.J. and Shoshani, A. "System Deadlocks" ACM Comp. Surveys 3, 2 (June 1971).

[5] Date, C.J. "An Introduction to Database Systems" Addison-Wesley, Reading, Mass. (Second Edition, 1977).

[6] Dijkstra, E.W. "Co-operating Sequential Processes" Programming languages: NATO advanced study institute. Editor: F. Genuys, Academic Press, London, 1968.

[7] Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. "The notions of consistency and predicate locks in a data base system" Comm. ACM 19, 11 (November 1976).

[8] Gray, J.N., Lorie, R.A. and Putzolu, G.R. "Granularity of locks in a shared data base" Proc. VLDB conference, Framlingham, Mass., 1975.

[9] Lorie, R.A. "Physical integrity in a large segmented database" ACM Trans. Database Syst. 2, 1 (March 1977).

[10] Palmer, I. "Database systems: A practical reference" CACI Inc. International, London, 1975.

[11] Rappaport, R.L. "File structure design to facilitate on-line instantaneous updating" Proc. ACM SIGMOD conference, 1975.

[12] Ries, D.R. and Stonebraker, M. "A study of the effects of locking granularity in a data base management system" ACM SIGMOD conference on management of data, Toronto, August 1977.

[13] Severance, D.G. and Lohman, G.M. "Differential files: their application to the maintenance of large databases" ACM Trans. Database Syst. 1, 3 (September 1976).

[14] - "ADABAS Introductory manual" Software AG, Hilberstrasse 20, 61 Darmstadt, W. Germany.

[15] - "TOTAL Users manual" Cincom Systems Inc.

[16] - "IMS/360 Utilities reference manual" IBM SH20-0915.