PUC

ON THE DESIGN OF A RELIABLE STORAGE COMPONENT FOR DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

by

Daniel A. Menascé

and

Oscar E. Landes

4/80

Pontifícia Universidade Católica do Rio de Janeiro

Technical Report DB 038002 - March 1980

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rua Marquês de São Vicente, 225 - CEP 22453

Rio de Janeiro — Brasil

On the Design of a Reliable Storage Component for Distributed Database Management Systems

Daniel A. Menascé

and

Oscar E. Landes

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, Brazil

Limited Distribution Notice

This report has been submitted for publication elsewhere and will probably be copyrighted if accepted for publication. It has been issued as a Technical Report for early dissemination of its contents. As a courtesy to the intended publisher, its distribution prior to publication should be limited to peer communications and specific requests.

1. Introduction

Robustness to failures must be kept in mind when designing any large software system. This is particularly true in the context of database management system design. Several important contributions to the area of crash recovery in database management systems appeared in the literature in the past few years [1,2,3,4 and 14]. Of these, the most relevant to the work reported here are the works of Lampson and Sturgis [1], Gray [2] and Lorie [3]. Verhofstaad in [5] contains a nice survey of crash recovery techniques.

Integrating several independently proposed ideas and crash recovery techniques into the design of a robust storage component of a distributed database management system (DDBMS) proved to be a non trivial task. The storage component is the portion of the DBMS responsible for maintaining the mapping of physical records into secondary storage and for the transfer of information between secondary storage and main memory. Lorie in [3] presents the design of a storage component of a centralized DBMS. His ideas could not be directly incorporated into our design since the storage component he proposed does not support the notion of a transaction as a sequence of actions which must be considered as an atomic operation. Nevertheless, we retained in our design some of the ideas proposed by Lorie. The notion of an intentions list, suggested by Lampson and Sturgis in [1], was also incorporated into our design.

Our design of a storage component of a DDBMS builds on several existing ideas and adds some other crash recovery strategies. The result is an integrated design which can lead to a direct implementation. This paper presents an informal but thorough description of our design. A formal, Pascal-like description can be found in [6]. The storage component proposed here is robust to transaction failures, system failures and secondary storage failures.

Section 2 presents the basic concepts and definitions used throughout the paper. Section 3 describes the organization of a DDBMS as a three component system - one of which is the storage component. The interface between this component and the remaining software is defined here. The next section describes the basic structure of the storage component, the update strategy it uses and the functions which compose its interface to the rest of the DDBMS. The paper concludes with a summary of performance analysis results of the storage component in terms of the price paid to achieve a robust design.

2. Basic Definitions and Concepts

The users of a database management system interact with it through transactions. A transaction is a sequence of actions of the type read, write (i.e. modify, insert or delete), lock and unlock. A transaction is considered the unit of consistency, in the sense that it takes the database from a consistent state into another consistent state [7]. A tran-

saction is also considered a unit of recovery, in the sense that either all or none of its actions must be reflected into the database [2], i.e. transactions must be considered as atomic operations. This property of a transaction must be enforced by the DBMS even in the presence of failures.

There may be several types of failures, but we will classify them into three categories according to the kind of recovery action that is needed: transaction failures, system failures and secondary storage failures. A transaction failure occurs when the normal termination of the transaction cannot be reached and the transaction must be aborted. A system failure is characterized by the loss of the contents of primary memory. Therefore, all in-progress transactions are lost. Secondary storage failures occur when part or all of the contents of the database is lost and must be recovered from a previously saved copy or dump.

It is important at this point to distinguish between the two types of memories that we consider:

i. volatile memory (main memory). It does not survive a system failure.

ii. non-volatile memory or stable storage (secondary storage). It usually survives system failures.

Stable storage is divided into fixed size blocks, called physical pages or simply pages, which are considered the unit of space allocation and the unit of transfer between main and secondary storage. We assume here that stable storage has the strong atomic property indicated in [1], i.e., the transfer

of a page from main memory into secondary storage may have either one of the following outcomes:

1. the page is not transferred
2. the page is successfully transferred

Each physical page stores the contents of a logical page. Logical pages may be allocated statically or dynamically. Static allocation means that a logical page is always stored in the same physical page even after the page is modified. Dynamic allocation implies that a logical page may be stored in any physical page at any point in time. In this case, there must be a pointer (stored in any physical page) which maps the logical page into the physical page which stores it.

There are two techniques to implement the strong atomic property depending on the type of allocation (dynamic or static) being used:

a. static allocation: in order to update a page, a free physical page is found and the modified original page is written into it. If this operation succeeds, then the contents of the new page must be written into the original page.

b. dynamic allocation: in order to update a page, a free physical page is found and the modified original page is written into the new page. If this operation is successful, the pointer which pointed to the original page must be updated to point to the new physical page.

In any case, the strong atomic property is achieved by associating two physical pages to a logical page during the update process. Figure 1 illustrates the two techniques above.

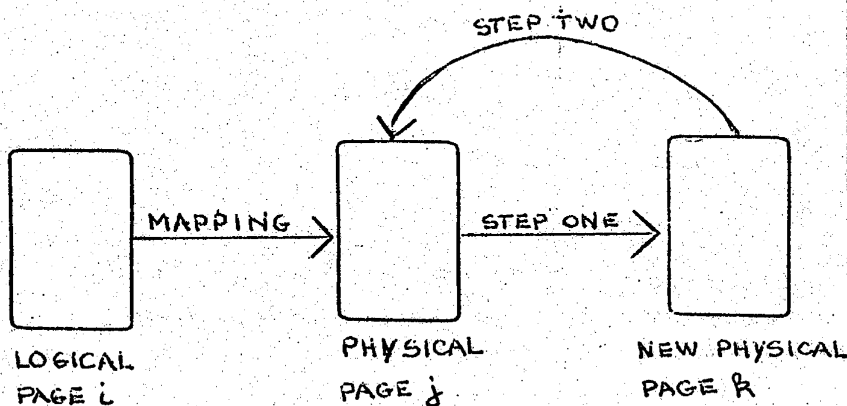

Figure 1.a - Implementation of Strong Atomic Property for Static Allocation

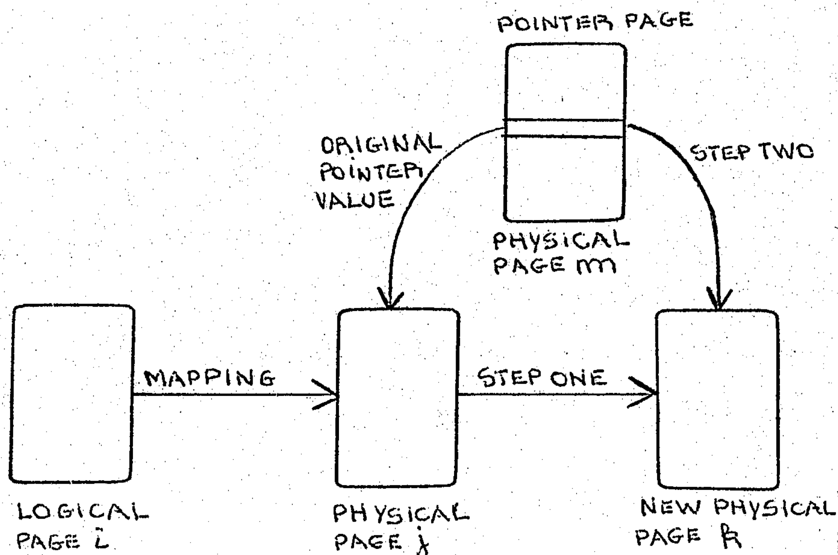

Figure 1.b - Implementation of Strong Atomic Property for Dynamic Allocation.

In figure 1.a, logical page i is always stored in physical page j. A new version of the logical page is written into a new physical page k. If the write operation on the physical page k is successful, page k is copied into the original page j. A failure may have the following effects on the values of physical pages j and k:

1. both pages contain their original values (the failure occurred before any of the write operations had started).

2. page k is detected to be in error and page j still has its original value (the failure occurred during the write of page k).

3. none of the pages are in error, but page j has the original value and page k has the new value of logical page i (the failure occurred immediately after the write of page k).

4. page k has the new value and the other has an incorrect value (the failure occurred during the write of page j).

5. both pages contain the new value of logical page i (the failure occurred after the write of page j).

In cases 1 and 2, the recovery procedure consists in restarting the update process from the first step. In cases 3 and 4, the recovery procedure consists in restarting the update process from the second step. Finally, in case 5, nothing has to be done since the update process had already finished when the failure occurred.

In figure 1.b, logical page i is initially stored in physical page j. There is a pointer page which implements this mapping. A new page k is used to store the contents of the modified logical page i. If the write operation on page k

is successful, physical page m (the pointer page) must be updated to reflect the new mapping.

If the pointer page is dynamically allocated (i.e., there is a pointer to it stored in a physical page) then we must repeat the procedure described in figure 1.b in order to update the pointer page. This recursiveness ends when we find a statically allocated page. Then we must use the procedure described in figure 1.a.

Let us now review the concept of <u>intentions list</u> introduced in 1 . An intentions list is a list of actions which must be executed in order to install into the database all updates of a given transaction. This list must be stored in stable storage with the strong atomic property. The intentions list must be idempotent, i.e., a single and complete execution of the list is equivalent (produces the same result) to the concatenated execution of partial lists followed by the execution of a complete list. For instance if L=A1,A2,A3,A4 is an intentions list, then A1,A2,A3,A4 is equivalent to A1,A2,A1,A2,A3,A1,A2,A3,A4.

In order for an intentions list to be idempotent, it is sufficient that each action of the list be of the form: write the value <u>v</u> into address <u>a</u>.

Note that once the DBMS succeeded in writing the intentions list in stable storage it is always possible to recover from a system failure. If the failure occurs while the intentions list is being executed, the recovery procedure consists

simply in restarting the execution of the intentions list from the begining.

In a distributed database environment, a transaction may update data stored at several sites. Therefore, there must be an intentions list at each of the participating sites. In order to implement atomic transactions, an intentions list can only start to be executed at a given site if all the sites have already written their intentions lists in stable storage. Lampson and Sturgis presented in [1] the protocol that must be followed by all the sites involved in the transaction in order to synchronize the execution of the intentions lists. This protocol has also been described by Gray [2] and called two-phase commit protocol. This protocol assumes that one of the sites involved acts as a coordinator. We assume here that the site of origin of the transaction is responsible for coordinating the execution of distributed transaction.

3. Internal Structure of a DDBMS

A database management system may be conceptually divided into three components:

a. high level component - responsible for creating and maintaining the user's views.

b. low level component - implements the logical to physical mapping.

c. <u>storage component</u> - responsible for the mapping of physical records into secondary storage and for the transfer of information from/into main memory.

This paper describes a storage component (SC) for a distributed database management system. This storage component is robust with respect to transaction failures, system failures and secondary storage failures. For the sake of simplicity, we will use the term DBMS to refer to the database management system except the storage component.

The interface between the storage component and the DBMS consists of a set of functions or operations implemented by the SC. The DB users submit transactions to the DBMS. The DBMS then translates the user commands into internal commands which include the functions implemented by the storage component. These functions will be described in section 4.3 and a complete example of a transaction and its corresponding translation appears in section 4.4.

The storage component assumes that the DBMS is in charge of the following tasks:

a. implement the concurrency control mechanism. Examples of such mechanisms can be found in [8,9 and 10]. It is important to note that the DBMS must control concurrency at the level of the user view of the database (logical level) as well as at the level of logical pages. In other words, transactions must hold any modified logical pages until end of transaction, which occurs when the intentions list of the transaction is created (see section 4.2).

b. handle deadlocks. Deadlock detection algorithms for distributed databases appear in [11].

c. detect any kind of failure and take the appropriate actions to restore the integrity of the DB. These actions may include calls to the functions provided by the SC.

d. maintain an audit trail.

e. recover the DB from the dump and the audit trail in case of secondary storage failures.

f. translate a sequence of user commands into a sequence of commands which include calls to the functions provided by the SC.

g. coordinate the distributed execution of a transaction.

h. implement the dumping policy.

i. implement Lampson & Sturgis protocol to coordinate the execution of a distributed transaction. The storage component provides some functions which are necessary for the implementation of this protocol.

4. The Storage Component

4.1 Basic Structure

The basic structure of the database considered here is similar to the one proposed by Lorie in [3] but it exhibits some important differences. The database is divided into segments. Each segment is a linear address space of variable size. Segments are divided into fixed size logical pages. Logical pages are stored in physical pages in secondary storage.

There may be K segments in each node. A logical page will be referenced by its sequence number within a given segment Sk, e.g., page i where i=1,...,Mk and Mk is the maximum

number of pages that may exist in segment Sk. Two consecutive logical pages of the same segment may be mapped into two non consecutive physical pages in secondary storage.

For each segment, there is a page table, which is a sequence of physical page addresses. The i-th element of a page table indicates the address of the physical page which is used to store the logical page in question. Each page table is implemented using physical pages (all physical pages are equivalent) called pointer pages. Figure 2 illustrates the basic structure of the database.

A page table may occupy more than one physical page. Therefore, a segment is composed of one or more pointer pages and several data pages. The pointer pages of each segment are numbered sequentially from one up.

In order to get access to the pointer pages which form a segment, the storage component maintains a segment table. There is an entry in the segment table for each segment of the local portion of the database. Each entry of this table contains a list of all the addresses of the physical pages which store the pointer pages which form the page table of the segment in question. Note that the i-th entry of this table corresponds to segment i. The j-th address of each entry corresponds to the j-th pointer page of the segment. The segment table is also implemented using physical pages which, in this case, are called segment pages.

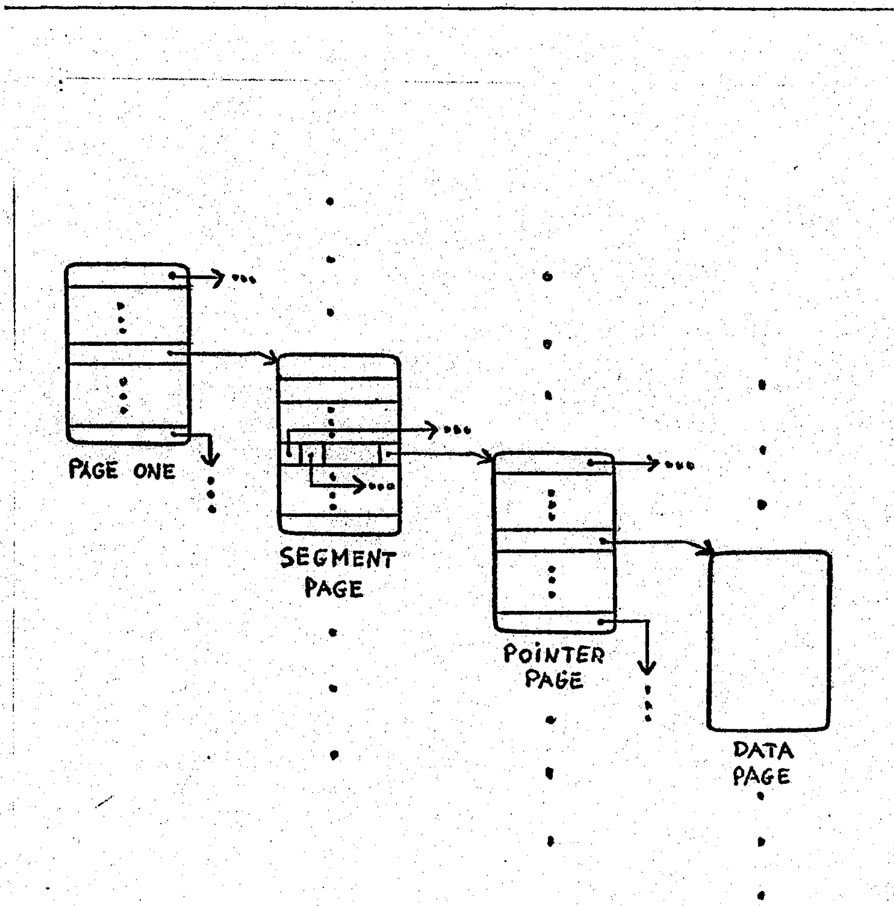

Figure 2 - Basic Structure of the Database

Finally there is a page in the system which contains a list of all the segment pages, in sequential order, of the local portion of the database. This page is called page one since it is statically allocated to physical page one. All the other pages in the system are dynamically allocated.

Segment pages, pointer pages and the page one will be called access pages as opposed to data pages which store the contents of the database.

In order to access logical page i of segment Sk, the storage component must execute the following steps:

1. Find the segment page which contains the entry corresponding to segment Sk. This page is pointed by the j-th entry in page one where j is given by the expression
$$j = \lfloor (k-1)/R \rfloor + 1$$
where R is the number of entries per segment page.

2. Find the pointer page which contains the pointer to logical page i. This pointer page is pointed by the n-th address in entry number (k mod R + 1) of the segment page found in the step above, where n is given by the expression
$$n = \lfloor (i-1)/N \rfloor + 1$$
where N is the number of data pages pointed by each pointer page.

3. Finally, the entry in the pointer page which points to the desired data page is the one numbered (i mod N + 1).

The storage component proposed here, uses a mechanism similar to that presented in [3] with respect to creation and modifications of data pages. The new versions of modified physical pages, as well as the first version of a new page, are built in free pages. In order to find these free pages the storage component maintains a bit array, in secondary storage, called MAP. The i-th entry of this array is one if the i-th physical page of the system is occupied and zero otherwise.

Some comments about the buffers used by the component are in order. All buffers are of the same size. Each buffer has associated with it the following information:

a. address in secondary storage of the physical page into which the contents of the buffer is to be transferred.

b. identification of the transaction which is using the buffer (only in the case of updates of data pages).

c. modification bit which indicates if the buffer must be transferred to secondary storage before it is reused.

d. lock counter: when a physical page if transferred into a buffer the transaction which originated the transfer locks the buffer. The buffer remains locked while the transaction is using it. In other words, the buffer management system cannot free a locked buffer. Notice that several transactions may be using the same buffer concurrently. As a result of this, the locking mechanism is implemented as follows.

To lock the buffer, the transaction increments the lock counter, associated with the buffer, by one. In order to free a buffer, the transaction must decrement the lock counter by one. In this way, the buffer management system will only reuse buffers which are not being used by any transaction, i.e., those with lock counter equal to zero. The lock counter is automatically incremented by the storage component when it delivers a buffer to the transaction. The transaction must explicitly decrement the lock counter through a function provided by the storage component (see section 4.3).

The SC maintains a per transaction list of buffers which contain data pages. Therefore, when a transaction is aborted, all buffers in the list for the transaction will have their lock counters decremented by one (see function ABORTR in section 4.3).

In summary, the database at each site is composed of the page one, segment pages, pointer pages, data pages and the pages which store the array MAP. All of them but the data pages are called acccess pages.

4.2 - Update Strategy

This section describes the update strategy used by the storage component to provide resilient update of the database. Access pages are always updated by actions of intentions lists. Recall that there is an intentions list associated with every transaction. The actions of these lists are updates to access pages only. Access pages always reflect a consistent state of the database. These pages are accessed concurrently by distinct transactions. Therefore, all updates to access pages should be seen by all transactions in order to avoid an inconsistency. For instance, consider the following situation: transaction T1 creates the segment page which is going to contain entries for segments 1 thru R and initializes the entry of segment 1. Concurrently, transaction T2 wants to initialize the entry for segment 2. This transaction cannot create a new page, but it must use the same page already created by transaction T1. This problem is solved as follows: the storage component maintains a copy of all access pages which are being updated. The copies are updated in place and, for each such update, the component creates actions in the proper intentions list to update the original access pages.

When the system is activated, a copy of page one is brought into main memory as well as a copy of all pages of the array MAP, which are stored in an array called CURRENT_MAP.

Figure 3 illustrates the update strategy by means of the following example. Consider an update on logical page i of segment k. As we already mentioned, data pages are updated using free physical pages to construct the new version of the data pages. The storage component must include two actions in the intentions list for every data page modified by a transaction. One to reset the bit in the array MAP corresponding to the physical page which contained the original version of the data page and another to set the bit in the array MAP corresponding to the physical page used to build the new version. In figure 3, this new version resides in physical page d. In order to reflect this new situation, the pointer page which corresponds to logical page i should be updated to point to the new data page.

Since the original pointer page is part of a consistent database state, the original pointer page is copied into a free page and the pointer to logical page i is updated in the copy to point to the new version of the data page. In order to update the original pointer page, an action must be included in the intentions list of the transaction in question. This action is of the form: "write the value d into word x of physical page b" where x is the word which contains the pointer to logical page i. Note that the original segment page which points to the original pointer page should be updated to point to the duplicated pointer page. At this point, the same process described above is repeated. In other words, the original segment page is copied into a free physical page and

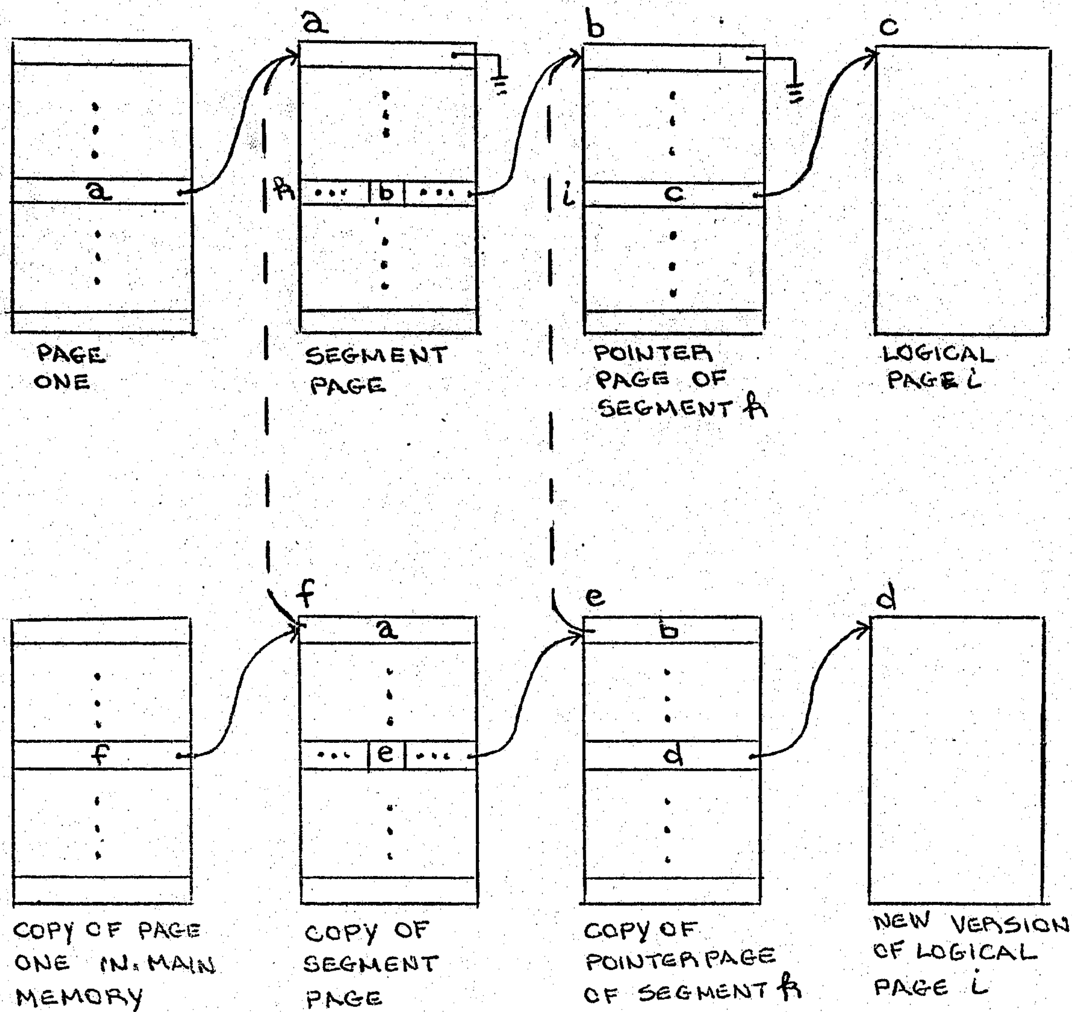

Figure 3 - Update Strategy

the address of the pointer page is updated in the copy. The address of the duplicated segment page should be updated in page one. Since a copy of page one always resides in main memory, this version of page one is updated to point to the duplicated segment page.

It is important to emphasize that after an access page has been duplicated once, it is not necessary to duplicate it again. All subsequent modifications can be done directly on the copies.

When an access page is duplicated, the corresponding bit in the array CURRENT_MAP is set to indicate that the physical page that stores the copy is not free.

An access to any data page is done through the copy of page one in main memory. Therefore, any transaction will always access the copies of the access pages instead of the original ones. This guarantees that all transactions will always see the current database state.

Periodically, the duplicated access pages should be freed. Care must be taken to preserve the database integrity. Therefore, the DBMS must make sure that there are no transactions in progress and that there are no intentions lists pending to be executed, since it is through the execution of intentions list that the updates are reflected into the original access pages. Let us define a <u>reorganization point</u> as the point in time when the above conditions are true and when the DBMS requests that the component frees all duplicated access pages. Therefore, duplicated access pages exist between two reorganization points. So, when an access page is to be updated, the component checks whether the page has already been duplicated since the last reorganization point. In the affir-

mative case the component updates the duplicated page, otherwise the component duplicates the page before updating it.

In order to introduce some other aspects of the update strategy, it is necessary to show, in more detail, the internal structure of an access page (segment or pointer page). As illustrated in figure 4, the first word of an access page contains a bit, called <u>create-bit</u>, the purpose of which will be explained later, and a pointer. This pointer is null for original access pages and it points to the corresponding original page in the case of duplicated access pages. By inspecting this pointer, the component is able to detect whether an access page is a copy or an original (see the dashed lines in figure 3).

The remaining words of an access page form a list of pointers to other access or data pages. Each such pointer is actually implemented as a pair (lock_bit,address). The lock_bit is used to indicate that the page pointed by the address part of the pointer is being duplicated by some transaction. This locking mechanism is necessary to avoid that two or more concurrent transactions duplicate more than once the same access page.

Let us now review the process of updating an access page P. Let P' be the access page which contains a pointer to P. The lock_bit in this pointer is set to one (if it is zero). Page P is accessed. If it is already a copy, then the lock_bit is set to zero, i.e., the page is unlocked. Otherwise,

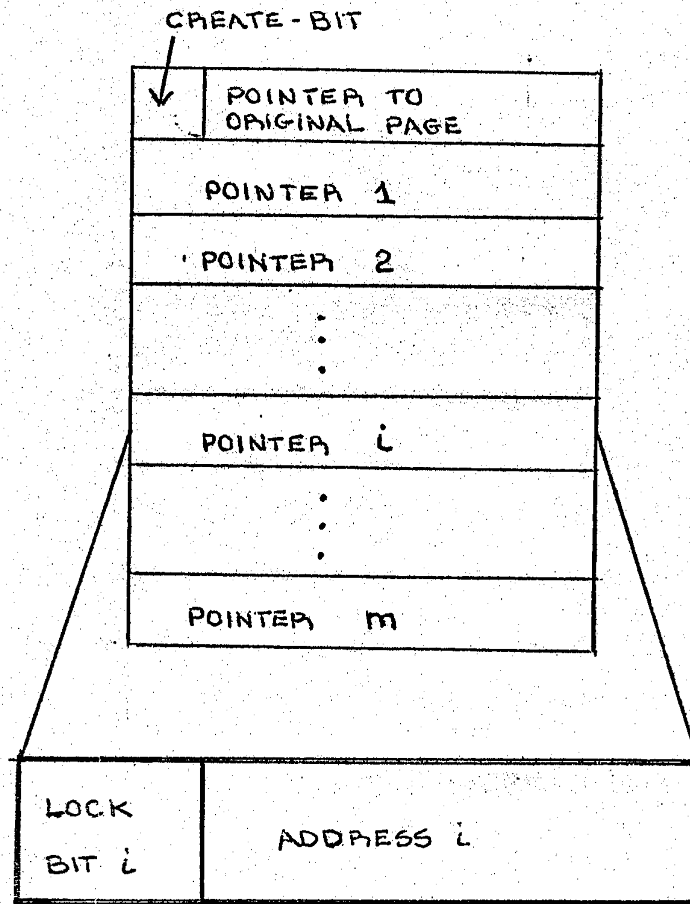

Figure 4 - Internal Structure os an Access Page

the page is duplicated, the pointer to P in P' is updated and page P is unlocked (lock_bit set to zero).

Let us now examine the use of the create_bit. Whenever a segment or pointer page is created, a free page is found, initialized with zeroes, and an action is included in the intentions list to update the pointer in the original page to point to the newly created page. Another action is included

in the intentions list to set the bit in the array MAP corresponding to the physical page used to store the access page being created. Besides, the page is duplicated as if it already existed. The copy is made to point to the original and the create_bit of the copy is set to one to indicate that the page is being created. The create_bit of original access pages is meaningless.

When a transaction T accesses a segment or pointer page which is being created by transaction T', the storage component must include in the intentions list of transaction T an action which updates the pointer to the page being created. This procedure ensures that even if transaction T' is aborted, all other transactions which used the access page are not affected. Whenever any of these transactions completes successfully, the database will reflect the creation of the new page.

The storage component must also include in the intentions list of every transaction which uses an access page that is being created, an action to reset the create_bit.

In order to free the duplicated access pages at a reorganization point, the DBMS must request that the component executes the function RESTART (see section 4.3). This function brings the original page one from secondary storage into main memory. In this way, the access to all the duplicated pages is lost. The function RESTART also copies the array MAP stored in secondary storage into the array CURRENT_MAP in

main memory in order to free the physical pages which store the duplicated pages. Notice that the array MAP never reflected the existence of these pages.

Recovery from system and transaction failures is straightforward given the update strategy described in this section. For instance, when a system failure occurs, the DBMS requests that the component executes the function RESTART. In this case, all the transactions which were in progress at the time of the failure must be executed again. If a transaction failure occurs, it is sufficient to delete the transaction identification from the table of active transactions since its intentions list has not yet been written into stable storage. It is important to recall that a transaction only updates the database when its intentions list is executed.

In summary, the pages which store the array MAP, all segment pages, pointer pages and page one are only updated by actions of intentions list. All copies of these pages are updated in place at transaction execution time.

4.3 - Functions Provided by the Storage Component

The high level and low level components of the DBMS must translate a transaction into a sequence of actions which can be executed directly by the storage component. This section describes the functions which are provided to the upper levels by the storage component. In the following description,

TID is the identification of the transaction which originated the call.

Segment Related Functions:

- CREATESEG(TID,SEG#): the storage component creates an entry for segment number SEG# in the segment table. All pointers to pointer pages of this segment are initialized as null. This function creates two actions in the intentions list of transaction TID. One to update the pointer in page one to the newly created segment page and the other to set the bit in array MAP corresponding to the physical page used to store the new segment page.

- DELETESEG(TID,SEG#): an action is created in the intentions list of the transaction to delete from the segment table, the entry corresponding to segment SEG#. All the pointer pages and data pages of this segment are freed. This is accomplished by creating actions in the intentions list to reset the appropriate bits of the array MAP.

Page Related Functions:

- READPAGE(TID,SEG#,PAGE#,BUFFADDR): the physical page which corresponds to logical page PAGE# of segment SEG# if transferred from secondary storage into a buffer. The address of this buffer is returned in BUFFADDR. The lock counter of the buffer is also incremented by the storage component.

- CREATEPAGE(TID,SEG#,PAGE#,BUFFADDR): the storage component finds a free physical page and associates its address to a buffer. The modification bit of the buffer is set to one, its lock counter is incremented and its address is returned in BUFFADDR. Before the buffer is reused, its contents will be transferred into secondary storage. The appropriate word in the appropriate pointer page of segment SEG# is made to point to the physical page just found, as a result of an action created in the intentions list of the transaction. An action is also created in the intentions list to set the bit in the array MAP corresponding to the physical page used to store the new logical page.

- DELETEPAGE(TID,SEG#,PAGE#): the logical page PAGE# is deleted from segment SEG#. An action is created in the intentions list to nullify the pointer to the data page in the appropriate pointer page. The physical page associated with the logical page is freed. This is accom-

plished by creating an action in the intentions list to reset the bit in the array MAP corresponding to the data page.

- UPDATEPAGE(TID,SEG#,PAGE#,BUFFADDR): the logical page PAGE# of segment SEG# is transferred to a buffer. A free physical page is found and its address is associated with the buffer. The modification bit of the buffer is set to one. Therefore, before the buffer is reused its contents are transferred to the free physical page associated with this buffer. The SC creates an action in the intentions list to update the appropriate pointer page of the segment SEG# to point to the new version of the data page. Actions are also created in the intentions list to reset the bit in the array MAP corresponding to the old version of the data page and to set the bit corresponding to the new one.

Whenever a data page buffer is delivered by the SC to the DBMS as in the functions READPAGE,CREATEPAGE and UPDATEPAGE, the SC includes the buffer in the list of data page buffers associated with the transaction.

<u>Buffer Related Function:</u>

- DECRBUFF(BUFFADDR): the lock counter of the buffer at address BUFFADDR is decremented by one, and the buffer is deleted from the list of data page buffers associated with the transaction.

Transaction Related Functions:

Since a transaction must be considered as an atomic operation, the actions of a transaction are delimited by a BEGINTR and by an ENDTR command.

- BEGINTR(TID): the SC creates an entry in the table of active transactions for transaction TID. The intentions list of the transaction is initialized as empty.
- ENDTR(TID): the intentions list of the transaction is created and written into stable storage. When this list is executed all updates generated by the transaction will be reflected into the database. The list will be executed upon demand of the DBMS through the use of the function EXECUTELST to be described shortly. As mentioned before, the DBMS implements Lampson and Sturgis protocol to synchronize the execution of the intentions list among all the participant sites.

Intentions List Related Functions:

Intentions list constitute the basic mechanism to implement atomic transactions. Actions of an intentions list include updates to segment tables, page tables, the page one and the array MAP.

When segment and page related functions are executed, actions of intentions lists are generated. As mentioned al-

ready the functions CREATESEG and DELETESEG generate actions to update segment tables and the array MAP. The functions CREATEPAGE, DELETEPAGE and UPDATEPAGE generate actions to update page tables and the array MAP.

Consistency of the database is preserved by the DBMS and the SC by following the rules below:

> Rule 1: the DBMS does not allow concurrent updates to logical pages.
>
> Rule 2: all resources locked by a transaction (including logical pages) are held until its intentions list is written in stable storage at all nodes involved in the transaction.
>
> Rule 3: at each site, intentions lists are executed in the order they were created. Notice that this restriction does not imply that transactions must be executed serially. As soon as an intentions list for transaction T is written in stable storage, its resources may be released and used by other transactions before the intentions list of T is executed.

As a result, when transaction T2 uses a logical page previously modified by a transaction T1, the system guarantees that the intentions list of T1 will be executed, at every node, before the one of T2.

Note that if Rule 1 were not enforced the SC would have to implement a concurrency control mechanism at the level of actions of intentions list, therefore increasing the complexity of the SC.

The level of concurrency can be increased by decreasing the size of logical pages at the expense of increasing the number of access pages. This is the classical space-time tradeoff.

Since the DBMS implements the concurrency control mechanism, it is also responsible for requesting that the SC executes intentions list in the proper order. The storage component provides functions to manipulate intentions lists.

Before an intentions list is created, the storage component must make sure that all buffers which contain modified data pages associated with the transaction have already been written into secondary storage. These buffers can be easily recognized due to the identification of the transaction associated to each data page buffer (see section 4.1). This strategy provides resiliency to system failures. Therefore, after an intentions list for a given transaction has been written into stable storage at all participant sites, it is always possible to recover from any type of crash.

. EXECUTELST(TID): the SC executes the intentions list of the transaction TID. After the execution, the list is destroyed and all the physical pages which store the list are freed. If a failure occurs during the execution of the intentions list, the DBMS must request the reexecution of the list.

. DELETELST(TID): if any of the participating sites is not able to write its intentions list, the DBMS coordinator must request that all intentions list already written be deleted. This is accomplished by this function.

Transaction Failure Related Function:

. ABORTR(TID): the SC deletes the entry associated with the transaction from the table of active transactions. All physical pages allocated to the transaction are freed. The list of buffers of data pages associated with the transaction is traversed and all lock counters of these buffers are decremented by one. All buffers which contain access pages used to obtain a given data page are locked and unlocked internally to the SC functions. Therefore, when an ABORTR function is being executed on behalf of transaction T, only data pages associated with T may be in buffers locked by T.

System Failure Related Function:

. RESTART(): the array MAP is copied into the array CURRENT_MAP, in order for it to represent the last state of integrity which existed before the failure. Page one must be brought into primary memory. Before any database activity can be restarted, all the intentions list already written must be executed. The DBMS must request that this be done, by invoking the function EXECUTELST. After all pending intentions lists are executed, the DBMS must request that the function RESTART be executed again, so that the last versions of the array MAP and of page one be brought into main memory.

Secondary Storage Failure Related Functions:

- START_DUMP(): the invokation of this function starts a dynamic dump process. The algorithm used here is the initial version method proposed by Rosenkrantz in [12].
- RELOADB(PAGELIST): the storage component receives a list of logical pages, PAGELIST, which have to be reloaded from the dump into the database. It is assumed that the DBMS implements the selective reload algorithm proposed by Menascé in [13].

4.4 - Example

Consider the following transaction in a banking system database:

"TRANSFER $100.00 FROM ACCT A INTO ACCT B"

The DBMS would translate this transaction, using SC functions, as indicated below:

```
BEGINTR(TID)

(assume that ACCT A is stored in page i of segment K)
UPDATEPAGE(TID,K,i,e1)

if ACCT A is valid
then begin

    balance(A) := balance(A) - 100
    if balance(A)    0
    then ABORTR(TID) ("negative balance")
    else begin

        (assume that ACCT B is stored in page j of
         segment L)
        UPDATEPAGE(TID,L,j,e2)
        if ACCT B is valid
        then balance(B) := balance(B) + 100
        else ABORTR(TID) ("ACCT B is invalid")
        DECRBUFF(e2)
        end
    end

else ABORTR(TID) ("ACCT A is invalid")

DECRBUFF(e1)
ENDTR(TID)
```

5. Performance Analysis Results

The average number of accesses to secondary storage per SC function as well as the ratio between the number of data pages and access pages appear in [6]. Due to space limitations we will only summarize some of the results in this section.

Table 1 shows the average number of accesses to secondary storage per function for the storage component proposed here and for a storage component which is not resilient to failures.

Let,

p1 = probability that a data page is modified more than once by the same transaction.

p2 = probability that a segment page has already been created. It is assumed that most of the data of the database has been entered. Therefore, this probability is fairly high ($\simeq .8$).

NPP = maximum number of pointer pages per segment.

C = number of physical pages which store the array MAP.

AVGMAP = average number of access pages modified by a transaction.

By inspection of Table 1, it can be seen that most of the overhead in terms of extra accesses to secondary storage lies in the execution of the function EXECUTELST. This is roughly the price that must be paid for a robust storage component.

FUNCTION		SC PROPOSED HERE	NON-CRASH RESILIENT SC
		AVERAGE NUMBER OF ACCESSES	
CREATESEG (p2=.8)		1.2	1.0
DELETESEG		2 + NPP/2	2 + NPP/2 + C
READPAGE	p1=.1	1.9	2.8
	p1=.5	1.5	2.0
	p1=.9	1.1	1.2
UPDATEPAGE	p1=.1	4.7	3.8
	p1=.5	3.5	3.0
	p1=.9	2.3	1.3
CREATEPAGE		4	3
DELETEPAGE		3	3
ENDTR		5	-
RESTART		C + 1	-
EXECUTELST		4+3*AVGMAP	-

Table 1 - Average Number of Accesses to Secondary Storage

The ratio, RDA, between the number of data pages and the number of access pages at a reorganization point is given by the following expression,

RDA = (N*K*NPP)/(2+K*NPP+(K-1)/R)

where,

N = number of pointers to data pages in a pointer page

K = number of segments of the database

R = number of pointers to pointer pages per segment page

Let us examine a numeric example to illustrate a typical value for RDA. Let N=1023, NPP=10. Therefore R= 1023/10 =102. The corresponding value of RDA is 1021.8 which is approximately equal to N. This is equivalent to saying that, for each access page in the system, there are N data pages. In this example, the overhead is of the order of 0.1%. Notice that the worst case happens when all access pages are duplicated. In this case the overhead will be 0.2%.

6. Conclusions

The storage component is the portion of the DBMS responsible for mapping physical records into secondary storage and for the transfer of information between primary memory and secondary storage. This paper presented an informal but complete design of a storage component for a distributed database management system which is robust to system, transaction and secondary storage failures. The update strategy used in the design of the component integrates some already existing techniques with some new ideas. The basic strategy consists in maintaining at any instant a consistent state of the database. This consistent state is modified only as a result of the execution of an intentions list. A current state of the DB also exists and is updated at transaction execution time.

The level of description presented here leads very easily to an implementation of the storage component.

The results of an analysis of the performance of the storage component indicate that the price to be paid to achieve a robust behavior is roughly proportional to the number of access pages updated by a transaction.

REFERENCES

[1] LAMPSON, B. and STURGIS, H.E., "Crash Recovery in a Distributed Data Storage System", Xerox Palo Alto Research Center, Palo Alto, California, USA, 1976 (to appear in the CACM).

[2] GRAY, J.N., "Notes on Data Base Operating Systems", Chapter 3.F. of the book "Operating Systems An Advanced Course", Springer-Verlag, 1978.

[3] LORIE, R.A., "Physical Integrity in a Large Segmented Database", ACM Transactions on Database Systems, Vol 2, No 1, March 1977.

[4] REUTER, A., "Minimizing the I/O - Operations for Undo-Logging in Database Systems", Proceedings of the Fifth International Conference on Very Large Data Bases, Rio de Janeiro, Brazil, October 3-5, 1979, pp. 164-172.

[5] VERHOFSTAD, J.S.M., "Recovery Techniques for Database Systems", Computer Surveys, Vol 10, No 2, June 1978.

[6] LANDES, O.E., "Recuperação de Falhas em Bancos de Dados Distribuídos: Especificação de um Componente de Armazenamento Confiável" ("Crash Recovery in Distributed Databases: Specification of a Reliable Storage Component"), Master Thesis, Computer Science Department, PUC/RJ, Rio de Janeiro, Brazil, January 1980.

[7] ESWARAN, K.P., GRAY, J.N., LORIE, R.A. and TRAIGER, I.L., "The Notions of Consistency and Predicate Locks in a Database System", Comunications of ACM, Vol 19, No 11, November 1976.

[8] MENASCÊ, D.A. et al., "A Locking Protocol for Resource Coordination in Distributed Databases, Proceedings of the 1978 ACM/SIGMOD Conference, Austin, Texas, May 31 - June 2, 1978 (to appear in the ACM TODS).

[9] BERNSTEIN, P. et al., "The Concurrency Control Mechanism of SDD-1: a System for Distributed Databases" (the general case), Tech. Rep CCA-77-09, Computer Corporation of America, Mass.,December 1977.

[10] STONEBRAKER, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978.

[11] MENASCÊ, D.A. and MUNTZ, R.R., "Locking and Deadlock Detection in Distributed Databases", IEEE Transactions on Software Engineering, May 1979, pp. 195-201.

[12] ROSENKRANTZ, D.J., "Dynamic Database Dumping", Proceedings of the 1978 ACM/SIGMOD Conference, Austin, Texas, May 31-June 2, 1978, pp. 3-8.

[13] MENASCE, D.A., "Selective Reloading of Very Large Databases", Proceedings of the Third International Computer Software and Applications Conference, Chicago, Illinois, November 6-8, 1979, pp. 583-587.

[14] GRAY, J.N. et al., "The Recovery Manager of a Dat Management System", IBM Research Laboratory Report RJ2623(33801), San Jose, California, August 15, 1979.