in: SFB-Bericht 26/88, SFB 124, University of Kaiserslautern, 1988

Buffer Management in a Non-Standard Database System

Andrea Sikeler
University Kaiserslautern

Abstract

Advanced computer applications (CAD/CAM, office automation, etc.) require complex data management functions which cannot be satisfied by currently available database systems (DBS). Therefore, so-called non-standard DBS (NDBS) attempt to satisfy these requirements by exploiting new architecture and implementation concepts. A prime concept is the DBS-kernel architecture which consists of an application-independent DBS-kernel and an application-oriented layer. Moreover, the DBS-kernel is divided into multiple layers representing different levels of abstraction. Each of these layers supports certain requirements of the non-standard applications needed at the chosen level of abstraction.

This paper concentrates on the storage system as the lowest layer in such a layered architecture. The requirements at this level of abstraction are outlined and an appropriate interface is proposed. New objects and operations as well as a transaction concept are introduced. Implementation aspects are discussed with respect to PRIMA, a prototype implementation of an DBS-kernel.

1. Introduction

Advanced computer applications such as office automation, geographical data processing or CAD/ CAM require complex data management functions which cannot be satisfied by currently available database systems (DBS) [Si80]. Therefore, so-called non-standard DBS (NDBS) are under development. They attempt to satisfy those requirements by exploiting new architecture and implementation concepts. A prime concept is the DBS-kernel architecture which consists of an application-independent DBS-kernel and an application-oriented layer. Numerous papers are published justifying the kernel architecture approach, its benefits, and its key properties [Da86, HR85, Pa87]. Our approach for an DBS-kernel named PRIMA (prototype implementation of the molecule-atom data model) is described in more detail in [Hä87].

PRIMA is divided into multiple layers representing different levels of abstraction (Fig. 1).Each of these layers offers at its interface a number of objects and operations supporting certain requirements of the non-standard applications needed at the chosen level of abstraction:

- The DBS-kernel, i.e. the data system [Sch88] as the highest layer, offers at its interface the molecule-atom data model (MAD model, [Mi88]) - an extension of the relational model, in particular a special form of the entity-relationship model. MAD supports direct and symmetric management of network structures as well as recursion. It provides language constructs to dynamically build complex objects (molecules) on top of basic objects (atoms).
- The interface of the access system [Si88a] is atom-oriented. Its expressiveness is similar to the interface of the Research Storage System (RSS, [As81]) of SYSTEM R [As76].

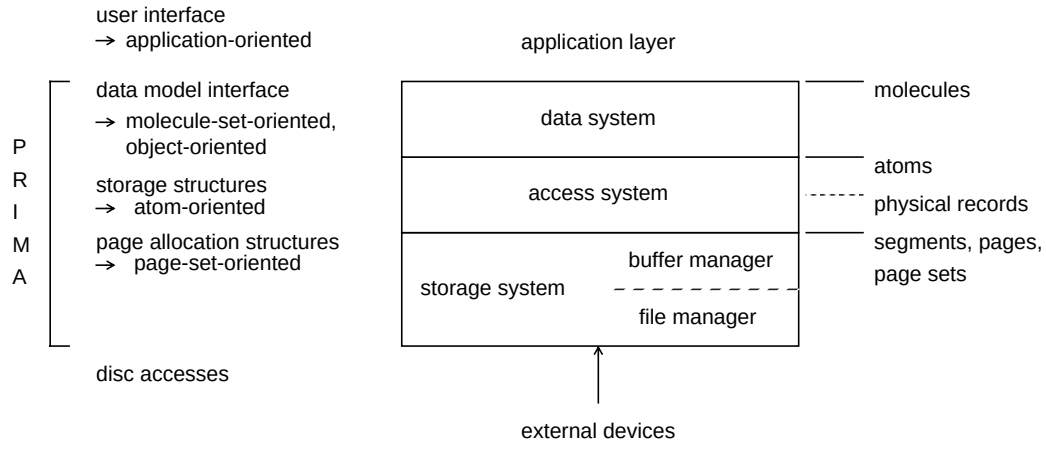


Figure 1: Architecture of a non-standard database system

- The storage system implements a set of "infinite" linear address spaces which allow for virtual addressing by the higher layers. Hence, the storage system of PRIMA represents a lower level of abstraction than RSS [As81], the Wisconsin Storage System [Ch85] or the Storage System of the DASDBS [DPS86].

This paper deals with the design and implementation of a storage system for an NDBS, especially for PRIMA. At first, the requirements for a storage system will be described in detail (chapter 2). These requirements are derived from the particular needs of applications to be supported and from the underlying hardware architecture available. Given the properties of the hardware architecture, it is useful to divide the storage system into a file manager (described in chapter 3) and a buffer manager. The design of the buffer manager is the central issue of the paper. In chapter 4, we describe the buffer-manager interface, i.e. the objects and operations at that interface, followed by an outline of the transaction concept (locking, logging, and recovery) supported by the buffer manager (chapter 5). Implementation aspects will be addressed in chapter 6. The paper concludes with a short summary and an outlook on current and future work.

2. Requirements for the Storage System

As in conventional DBS [Hä78] the storage system in PRIMA pursues two major tasks:

- First, the storage system has to manage the database buffer. Thus, higher layers, i.e. the access system, can manipulate the contents of the database with simple machine instructions (such as COMPARE, MOVE, etc.).
- Second, the storage system has to organize the external storage devices (disks) for the entire database. Hence, it is responsible for the exchange of data between main storage and disk storage.

In order to facilitate this data exchange, the database is commonly divided into pages of equal size (generally 512 to 4096 bytes). Typically, the database buffer consists of page frames of the same size [EH84]. So, the unit of data exchange is usually one page requested on demand. As a consequence, in most conventional DBS (e.g. SYSTEM R [As76], UDS [UDS], etc.) the size of the objects manipulated by the access system (records or tuples) is limited by the page size of the storage system.

Thus, in these systems modeling of application objects to be stored in the database must consider the page size. In commercial applications objects are simple, they can be described by a single record of limited size (approximately less than 2000 bytes). The objects of non-standard applications, however, are generally more complex and often composed of other simple or complex objects (e.g. the boundary representation of a solid [BB84, Mi85]). Even a simple object can be described by only a few bytes (such as a point in CAD/CAM or geographic data processing) or by some Mbytes (such as an image in image processing). Hence, all new data models for NDBS (NF2 [SS86], extended relational model [Lo84], or MAD [Mi88]) include a data type LONG FIELD, BYTE VAR, etc. for very long attributes. Furthermore, all these models offer the possibility to build complex objects, either in a static or dynamic manner. As a consequence, such concepts must be supported both by the access system and by the storage system. The access system should be able to handle records spanning two or even more pages. It should cluster all records describing one complex object into one or more pages [DPS86, HR85]. For that purpose, it should be possible to define a set of pages. These page sets should be treated by the storage system as a single page, i.e. the page set as a whole is transferred from and to the database buffer (e.g. by chained I/O). In addition, it seems useful to support different page sizes. Then the page size may be defined in order to approximate the record size, and many synchronization conflicts at the page level can be avoided. Furthermore, pages have to be created and deleted dynamically to adapt the number of pages to the current amount of data, thus, allowing for dynamic growth and shrinking of segments.

These three requirements for the storage system (dynamic creation and deletion of pages, different page sizes, and page sets) apply to all NDBS (DASDBS [Pa84], etc.). However, two additional requirements for the PRIMA storage system result from special characteristics of the PRIMA access system [Si88a]:

- It should be possible not only to make a predefined page set but also an arbitrary set of pages available by one storage system operation. Such a feature, for instance, greatly simplifies the construction of complex objects in a dynamic manner.
- he access system may redundantly store a simple object (record) in several pages. For retrieval, any of these pages may be selected. Hence, the storage system can choose the page with minimum cost (considering data exchange and synchronization needs).

Summarizing the above introduced concepts, the following main requirements for the PRIMA storage system can be stated:

- dynamic creation and deletion of pages
- different page sizes
- predefined page sets supported by physical clustering
- arbitrary page sets
- selection of an arbitrary page from a page set.

In order to satisfy these requirements the storage system is divided into a file manager organizing the external devices and into a buffer manager maintaining the database buffer. In PRIMA, the file manager of the underlying operating system is used (chapter 3) which manages its own file "cache". As in conventional DBS, this cache cannot be used directly as database buffer for different reasons [CHMS87, EH84, St81]:

- Access overhead is too high.
- Replacement decisions in the file cache may not be optimal.
- Selective flushing of pages to disk is not possible.

In PRIMA, a critical source of the high access overhead especially results from the underlying hardware architecture (Fig. 2) using loosely coupled processors [HR86]. Only one of these processors has access to external devices, thus, the file manager has to run on this processor. The application-oriented layer will also run on a single processor, while the remaining part of PRIMA, i.e. the data system and the access system, is intended to run on multiple processors to exploit parallelism for database management work. To simulate the infinite linear address spaces for the access system, each of these processors has to have a buffer manager, responsible for the corresponding database buffer. Unfortunately, this solution leads to a buffer invalidation problem known from the field of DB-sharing systems [Ra86]: Multiple copies of a database page may reside in different buffers simultaneously. Thus, modification of any copy invalidates all other copies. Since access to obsolete data has to be avoided, modified pages have to be exchanged among processors. There are many solutions to the buffer invalidation problem in DB-sharing systems [Ra86] which are also applicable for PRIMA. The one chosen for PRIMA will be described in chapter 4.

3. The File Manager

The file manager used in PRIMA is part of the underlying distributed operating system DISTOS, developed at the University of Kaiserslautern [Ne87]. It offers operations on files, blocks, and clusters at its interface [Fr87].

Operations on Files

The operations on files comprise of means to create or delete a file, to open or close a file, and to expand or shrink a file. When creating a file one has to specify a file name, a block size, a number of blocks for

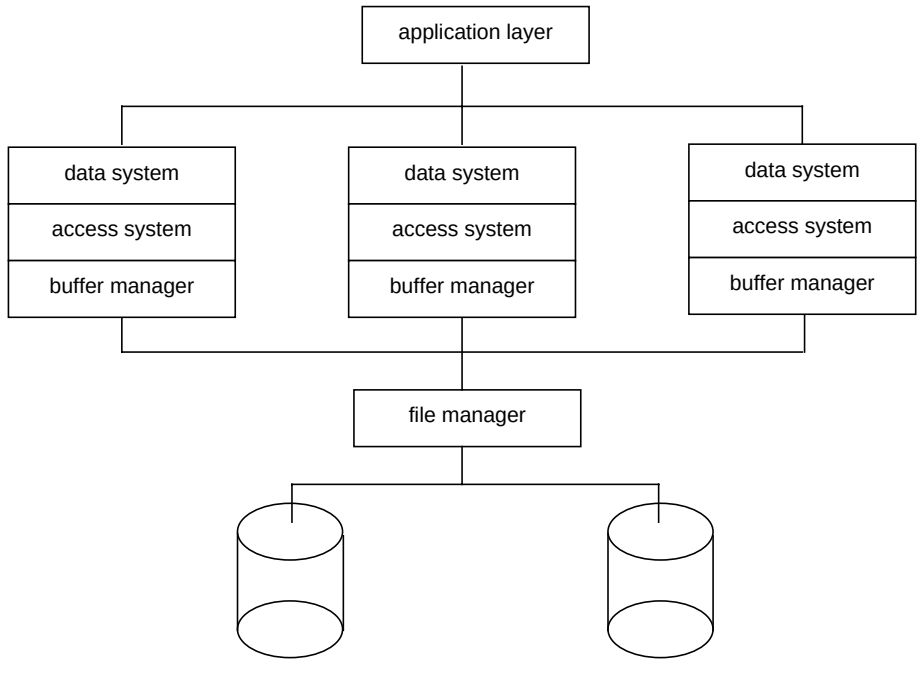


Figure 2: Hardware architecture

first allocation, and a disk group. At present, five different block sizes are available: 1/2, 1, 2, 4 and 8 kbytes. The block size is kept fixed during the lifetime of a file. The parameter ’disk group’ allows for separate storage of files on different external devices. This feature can be used for example to separate the log file and the original file on different disks. The specified number of blocks are allocated and logically numbered starting with block number 1. An unique file identifier is assigned to the created file. This file identifier has to be used by most of the operations to identify the current file. To expand a file, the file manager offers two different operations: "create-blocks" allocates new blocks without changing the logical block numbers of existing blocks. The logical block numbers of deleted blocks are reused or the new blocks are appended at the end of the file. If consecutive logical block numbers are needed, the parameter ’consecutive’ must be set ’true’. The operation "insert-blocks", however, allocates new blocks by inserting blocks after a specified block. The logical block numbers of existing blocks may be changed. When deleting blocks, it is possible to delete a sequence of blocks starting by the specified block number or to delete individual blocks identified by their block number. The block numbers of other blocks are not modified. To prevent gaps, the file may be renumbered. An additional operation is to copy blocks from one file to another.

Operations on Blocks

The file manager supports direct access to the blocks of a file. It is possible to read as well as to write an arbitrary number of single blocks identified by their block number, a sequence of blocks starting by a specified block number, and the whole file. The "write" operation may allocate new blocks automatically if necessary. Thus, the operations on blocks are set-oriented.

Operations on Clusters

The logical numbering of blocks of a file does not correspond to the physical ordering on disk since the blocks of a file may be spread over the whole disk. This may result in an inefficient access to a set of blocks. Hence, the file manager offers ways to cluster blocks. All blocks belonging to a cluster are stored on disk in such a way that an efficient access to the whole cluster is feasible, e.g. by chained I/O. The operations on clusters are to create or delete a cluster, to insert blocks into a cluster or to remove blocks from a cluster, and to read a cluster. A special operation for writing a cluster does not exist. In the current implementation of the file manager writing a cluster corresponds to writing a set of blocks.

4. The Interface of the Buffer Manager

Based on the objects and operations of the above introduced file manager the buffer manager provides a number of "infinite" linear address spaces by managing a database buffer. As in conventional DBS, the database is divided into various segments consisting of a set of pages. Additionally, page sequences are introduced as a special cluster mechanism within a segment. In the following, each of these objects together with the corresponding operations are described in more detail.

4.1 Segments

Segments are linear address spaces consisting of a set of logically ordered pages. Each segment can be described by a set of parameters which establish its characteristics:

segment number and segment name

Both segment number and segment name are used for identification purposes. The segment name is set by the "user", i.e. the access system, whereas the segment number is set by the buffer manager itself.

page size

The page size of a segment can be 1/2, 1, 2, 4 or 8 kbytes depending on the block sizes of the underlying file manager. All pages of a segment are of equal size which is kept fixed during the lifetime of the segment.

segment type

The segment type encodes additional characteristics of a segment, each describing it in more detail [Hä78]:

- The lifetime of a segment distinguishes between permanent segments as part of the database and temporary segments as working storage. The latter are deleted automatically, either by shutdown of PRIMA or at the end of the current transaction, i.e. the transaction creating the temporary segment. Whether both possibilities are necessary depends on the requirements of the access system. At present both are supported.
- The kind of usage distinguishes between public and private segments. A public segment can be used concurrently by all users of the buffer manager whereas a private segment can be used only by one user at a time. A user corresponds to a transaction at the level of the buffer manager (chapter 5).

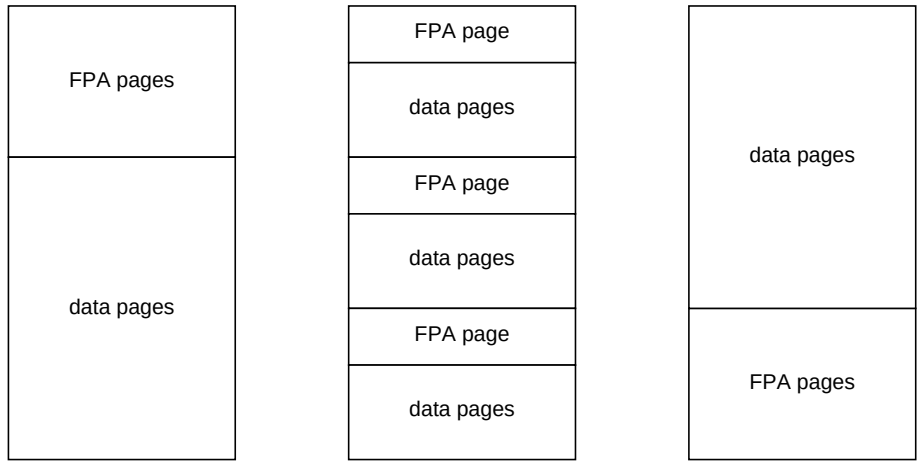


Figure 3: Grouping of the FPA pages

- The third characteristic affects the recovery mechanism used in the case of a failure (system crash, transaction abort, etc.). Recovery can be performed automatically by the system, i.e. the buffer manager (chapter 5), or explicitly by the user, i.e. the access system. Whether both features are necessary depends on the recovery concept of the higher layers. Therefore, first the buffer manager only supports automatic recovery of all segments.

These three characteristics are also used in other DBS (e.g. SYSTEM R [Hä78]) to build segment types. The fourth characteristic, however, is characteristic for PRIMA. In PRIMA, the buffer manager includes the segment-internal free place administration (FPA). For that purpose, each segment contains a number of so-called FPA pages, each containing the free place information for a fixed number of "normal" data pages. In other words, the FPA pages store an FPA entry for each data page of the segment indicating the number of free bytes for the page. Within a segment the FPA pages can be located in different ways (Fig. 3):

1. All FPA pages are located at the beginning of the segment. This approach is only useful, if segments are not dynamic or, alternatively, if all necessary FPA pages can be located at creation time of the segment. Otherwise, expanding or shrinking of the segment implies relocation of data.
2. The FPA pages are distributed over the segment in such a way that every FPA page is followed by n data pages. Hence, the current number of FPA pages can be adapted exactly to the current size of the segment. As a consequence, address space mapping becomes more complicated. Data structures and algorithms such as database key translation tables or hashing, which need consecutive page numbers, have to consider the intermediate FPA pages.
3. All FPA pages are located at the end of the segment. The current number of FPA pages can also be adapted to the current size of the segment. Furthermore, all data pages receive consecutive page numbers. However, the contents of the FPA pages have to be moved or the corresponding pages have to be renamed when a segment grows or shrinks.

- Thus, the fourth characteristic encoded in the segment type describes the way to locate FPA pages. The first of the above ideas does not seem appropriate. Therefore, only alternatives two and three are considered.

All parameters of a segment are summarized in Fig. 4.

Operations on Segments

parameter	possible values
segment number	
segment name	
page size	512 bytes 1024 bytes 2048 bytes 4096 bytes 8192 bytes
segment type	
lifetime	permanent temporary - end of transaction temporary - system shutdown
kind of usage	private public
recovery mechanism	automatic
grouping of FPA pages	spread over the segment consecutive at the end of the segment

Figure 4: Parameters of a segment

First of all a segment can be initialized. The segment and the corresponding file are created and the necessary FPA pages are initialized. The parameters of INIT-SEGMENT correspond to the parameters of a segment, described above, plus a number for first page allocation. All parameters are input parameters except the segment number which is set by the buffer manager. The number for first page allocation corresponds to the number of data pages. The corresponding number of FPA pages is determined by the buffer manager, and the FPA entries of all data pages are initialized with the maximum number of free bytes depending on the page size. All segment parameters needed for further operations on this segment are stored as meta information. This meta information is managed together with the meta information of the other layers by a special component [We87].

The operation DELETE-SEGMENT releases the corresponding file and deletes the meta information of the segment. The segment is identified by the segment number.

In NDBS segments should be dynamic. This is achieved by the operation GET-FREE-PLACE that automatically allocates new pages if the requested number of free bytes cannot be satisfied by existing pages. The operation either returns a single page number or a set of page numbers depending on whether the requested number of free bytes can be satisfied within one page (i.e. request number of free bytes £ maximum number of free bytes within a page). If the segment in question is "consecutive", i.e. the FPA pages are located at the end of the segment, then the returned set of pages build a logical continuous address space. Otherwise, the returned pages are spread over the whole segment. If a set of page numbers is returned the corresponding data pages are empty. Otherwise, the page may contain data.

To determine the page number returned by GET-FREE-PLACE the FPA entries are used. Hence, the free place information in the FPA entries is always assumed to be up-to-date. There are different operations to modify the FPA entry of a certain page. MODIFY-FPA overwrites the existing value whereas INCREMENT-FPA and DECREMENT-FPA alter the existing value by a specified value. When a data page becomes empty, i.e. the FPA entry gets a value equal to the maximum number of free bytes for the corresponding page size, that page can be deleted. In order to avoid separate management of deleted pages, pages are only deleted if they reside at the end of a segment. All other empty pages within a segment are managed by the free place administration, they are reused by GET-FREE-PLACE. In the case of a

consecutive segment, deleting a single page causes some overhead for rearranging the FPA pages. Therefore, empty pages are deleted only if there is a certain number of pages to be deleted.

Segments are opened implicitly by the first operation on them, and they are closed implicitly by system shutdown. Therefore, no corresponding operations are offered at the buffer-manager interface.

4.2 Pages

As already mentioned, segments are divided into pages of equal size. Each of these pages is identified by a page number unique within the segment. Two different page types can be distinguished: FPA pages and data pages. FPA pages are used only for free place administration (FPA). As the FPA is integrated into the buffer manager these pages are not available outside the buffer manager. Data pages, however, are used by the access system to store "user objects", i.e. physical records, address information, access path entries, and so on. Therefore, the contents of an FPA page are handled by the buffer manager, whereas the contents of a data page are unknown to it with the exception of the page header and the page-internal free place information. The page header as well as the free place information are initialized by the buffer manager along with the first access to a page. Furthermore, the page header is used for error detection [Kü85] by checking the page header whether the right page was delivered by the file manager. For this purpose, each page header contains the segment number and the page number of the corresponding page (Fig. 5). Using the page type the buffer manager can prevent FPA pages from being read by the access system. The page size as well as the remaining part of the page header, however, are used by the access system. The page internal free place information consists of the current number of free bytes within the page as well as the start address of the free place (Fig. 5). These two values are initialized by the buffer manager and maintained by the access system.

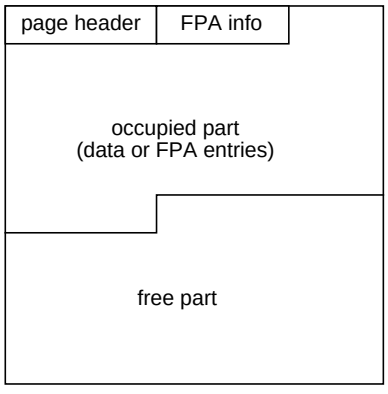
Operations on Pages

The FIX-PAGE operation asks the buffer manager to locate the corresponding page in the database buffer or to fetch it from external storage if it is not yet in the buffer, and to fix it in the buffer. Now the access system can execute machine instructions on "user objects" stored in the page up to a corresponding UNFIX-PAGE operation.

The parameters of the FIX-PAGE operation are the segment number and the page number, which are used for identification purposes, and the fix mode. The fix mode indicates whether the access system wants to read or to write the requested page. Moreover, it is used to lock the page in the corresponding mode (chapter 5). As a result of the FIX-PAGE operation the address of the page in the database buffer is returned.

The parameters of the UNFIX-PAGE operation are the segment number and the page number as well as a CHANGED bit and an UNLOCK bit. The CHANGED bit indicates whether or not a page with fix mode WRITE was actually modified. If not, the page does not need to be written back to the database. The UNLOCK bit indicates that the corresponding page lock can be released before the end of transaction (chapter 5). This feature is mandatory for some parallel algorithms on access path structures [Sh84].

The actions performed by the buffer manager for a FIX-PAGE and an UNFIX-PAGE operation as well as the underlying transaction concept are described in more detail in chapter 5.




```
page header     = record
                      segment number          (2 byte)
                      page number             (3 byte)
                      page size               (2 byte)
                      page type               (1 byte)
                      virtual page number     (3 byte)
                      not used                (5 byte)
                  end

FPA info        = record
                      number of free bytes    (2 byte)
                      beginning of free part  (2 byte)
                  end
```

Figure 5: Internal structure of a page

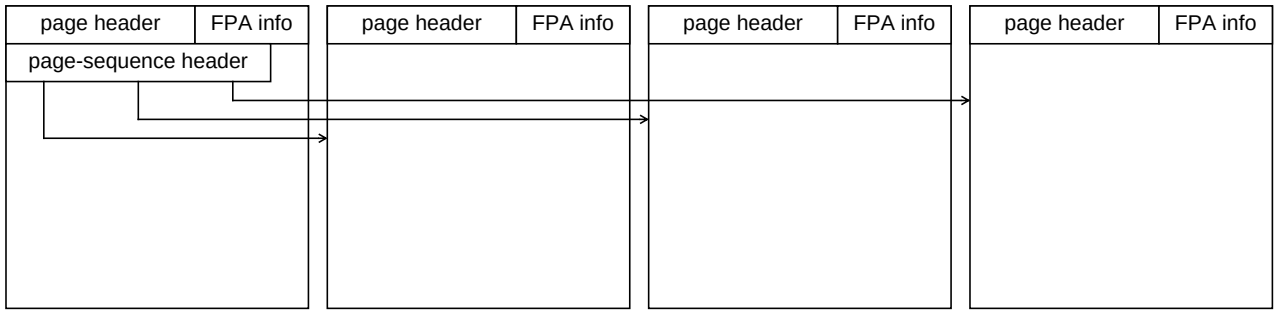
4.3 Page Sequences

One of the demands on the buffer management in an NDBS is a mechanism to define a page set supported by physical clustering (chapter 2). To satisfy this requirement page sequences are introduced. A page sequence is a set of logical consecutive pages of a segment which contain (from the view-point of the access system) one single "user object" spanning these pages. A page sequence consists of a so-called header page and one or more component pages (Fig. 6). The header page contains the general page header and the free place information as well as a so-called page-sequence header. This page-sequence header is a variable-length list of all pages (page numbers) belonging to the page sequence. Furthermore, the header page may contain a part of the "user object". To support physical clustering a page sequence corresponds to a cluster at the file level (chapter 3).

Operations on Page Sequences

Page sequences must be created, expanded, reduced, and deleted by special operations. When creating a page sequence the header page and the component pages have to be specified. Each page sequence is assigned a page-sequence number identifying the page sequence within the segment. This page-sequence number corresponds to the page number of the header page. Hence, the header page cannot be removed from the page sequence. When reducing the page sequence it is automatically deleted when the number of component pages becomes zero. Therefore, the management of the page-sequence header is integrated into the buffer manager. If the page-sequence header cannot be modified because there is not enough free place within the header page, the corresponding operation (create or expand) will be rejected. Furthermore, the page-sequence number is added to the page header (Fig. 5) of each page belonging to a page sequence. Therefore, it is easy to check that a page can only belong to exactly one page sequence.

To provide a page sequence the operations FIX-PAGE-SEQUENCE and UNFIX-PAGE-SEQUENCE are used. For a FIX-PAGE-SEQUENCE operation the buffer manager has to fix the whole page sequence in the database buffer, i.e. all pages belonging to the page sequence. The pages of the page sequence are distributed over the database buffer. That means, the page sequence does not build one contiguous linear address space within the buffer, and the access system must be aware of the page boundaries. As the result of the FIX-PAGE-SEQUENCE operation the buffer manager returns the address of the header page and the addresses of the component pages. An UNFIX-PAGE-SEQUENCE operation releases all pages belonging to the corresponding page sequence. Using the UNCHANGED bit for each of the pages the access system can indicate which of the pages were actually modified.




```
page-sequence header      = record
                              number of pages                              (1 byte)
                              array [1 .. number of pages] of page number  (3 byte per page)
                            end
```

Figure 6: Page sequence

4.4 Page Sets and Page Contests

Since not only predefined page sets are required by the access system, the buffer manager offers three operations to handle an arbitrary page set. The first operation FIX-PAGE-SET fixes all specified pages and page sequences in the database buffer. In other words, this operation is just a shorthand for a number of subsequent FIX-PAGE/FIX-PAGE-SEQUENCE operations. As an additional parameter the PAGE-TYPE bit indicates for each page number whether it addresses a page or a page sequence.

Correspondingly, the second operation FIX-ONE-PAGE fixes only one of the specified pages or page sequences. Therefore, the buffer manager selects the page (page sequence) that causes minimum costs, i.e. the buffer manager performs a "page contest". The selection criteria are:

- Pages (page sequences) are not locked by another transaction.
- Pages (page sequences) are already located in the database buffer.
- Pages are cheaper than page sequences.
- Small pages are less expensive than larger ones.

The third operation UNFIX-PAGE-SET corresponds to the FIX-PAGE-SET operation. All specified pages and page sequences are released and the corresponding CHANGED and UNLOCK bits are evaluated.

So far, the objects and operations at the buffer-manager interface have been outlined. In the next chapter, the underlying transaction concept (locking, logging, and recovery) is discussed, which integrates the operations BEGIN-OF-TRANSACTION, END-OF-TRANSACTION, and ABORT-TRANSACTION into the buffer manager. Furthermore, the actions performed by some of the above operations are described in more detail.

5. The Transaction Concept

The operations at the buffer manager interface are embedded in a transaction concept [CP88] following the ACID principle [HR83]:

- atomicity: Either all or none of the modified pages are in the database.
- consistency: Consistency depends on the assumption that each transaction reaching its normal end does only legal things. That means, the committed results of a transaction always preserve the consistency of the database.
- isolation: The modifications of a transaction are independent of all other transactions running in parallel.
- durability: All modifications of a successfully finished transaction survive any subsequent malfunction (system crash etc.).

The mechanisms used in the buffer manager to guarantee these properties will be described in the following sections. For that purpose, it is necessary to explain the actions performed by the FIX and UNFIX operations in more detail.

5.1 Details of the FIX and UNFIX Operations

The actions which the buffer manager has to carry out are always the same for all kinds of FIX operations (FIX-PAGE, FIX-PAGE-SEQUENCE, etc.) [EH84]:

- The buffer is searched for the page, the page sequence, or the page set and the corresponding page or pages are located.
- If the page (pages) is not in the buffer, a replacement algorithm together with a buffer allocation policy determines a buffer frame for the page. "Old" pages in that buffer frames have to be replaced. All previously modified pages selected for replacement have to be written back to disk before the requested page (pages) can be located in the buffer frame.
- The requested page (pages) is fixed in the buffer and the address of the buffer frame (frames) is returned to the access system.

There are different search strategies to determine whether a requested page is already in the buffer [EH84]. One of the most efficient is searching via a hash table. The hash algorithm transforms a segment number and a page number into a displacement in a page table where an entry describing the page and its current position in the buffer can be found (Fig. 7). The page table is divided into a primary part containing "home" entries and a secondary part containing overflow entries chained to their corresponding "home" entry.

In conventional DBS the current position of a page corresponds to the number of the buffer frame where the page can be found. To minimize buffer fragmentation, all database pages are of equal size, the database buffer consists of frames of the same size, and each page occupies exactly one frame. Thus, only one page has to be replaced. The PRIMA buffer manager, however, supports pages of different size. Hence, either the database buffer has to be organized in another way or the replacement algorithm has to be modified in such a way that it considers different page sizes.

A special organization of the database buffer is to divide it into different partitions according to the number of possible page sizes or frame sizes [Pa84]. To restrict the number of partitions a significant number of page sizes should be selected. Hence, the same buffer allocation policies and replacement algorithms as in conventional systems can be used for each partition. The only problem left is to define the "optimal" size of each partition for minimizing the global buffer fault rate. As this depends on the current work load, it seems useful to start with predefined partition sizes and to adapt them dynamically.

Another way to deal with different page sizes is to modify a well-known replacement algorithm or to design a new one. In PRIMA, the first possibility was chosen. The new algorithm VAR-PAGE-LRU [Si88b] is based on the well-known LRU (least recently used) algorithm. For simplification we assume a global buffer allocation strategy, i.e. all pages which are not fixed at replacement time can be taken into account by the replacement algorithm. A combination of VAR-PAGE-LRU with other buffer allocation policies [EH84] (transaction oriented or page-type oriented) is not planned for the moment but seems to be very interesting for future research.

VAR-PAGE-LRU [Mi86, Si88b]

VAR-PAGE-LRU was designed for a (database) buffer divided into frames of equal size and for pages of different sizes which are a multiple of the frame size. It is useful to restrict the page size to the one-, two-, fourfold etc. of the frame size to avoid serious fragmentation of the buffer. Hence, in PRIMA a frame size of 512 bytes (the smallest block size offered by the file manager) and page sizes of 1/2, 1, 2, 4 and 8 kbytes were chosen in order to store each page in multiple consecutive frames.

The data structures used by VAR-PAGE-LRU are an LRU-chain and a so-called free-list. The LRU-chain contains all entries of the page table where the corresponding page is unfixed. The entries are sorted according to the time of the last unfix operation on the page. So, "least recently used" is interpreted as "least recently unfixed" and not as "least recently referenced" [EH84]. The beginning and the end of the LRU-chain are indicated by LRU-FIRST and LRU-LAST, respectively. The free-list is a bitlist containing one bit for each buffer frame. This bit indicates whether the corresponding frame is free ("1") or occupied ("0"). Occupied means that the frame itself or a consecutive number of frames it belongs to represents a page with a corresponding entry in the page table, whereas free means that the frame contains a part of a previously replaced page. Free buffer frames may result from a page being replaced by a smaller one.

So, the first step of VAR-PAGE-LRU is to search the free-list for a number of consecutive free frames sufficient for the requested page. If such a sequence exists no page has to be replaced. Otherwise, the replacement algorithm starts to determine one or more consecutive pages within the buffer which then have to be replaced to get enough free place (perhaps in combination with free buffer frames) for the requested page. The algorithm works in two steps:

1. Searching for a sufficient range of free frames (Fig. 8)

In a first step the LRU-chain is searched, starting with the "oldest" page, until enough consecutive pages are found to satisfy the request. For this purpose, a second bitlist, the so-called work-list, is necessary. This work-list is used to simulate the behaviour of the free-list by testing the consequences of freeing more than one page before actually doing so. Hence, the work-list is initialized by the current free-list. Then the LRU-chain is searched. For each page it has to be examined whether the frames occupied by the page, perhaps together with neighbouring free frames from the free-list, are sufficient for the requested page. In this case, it is the only page to be replaced, and the algorithm stops. The same check is done with the work-list, and the algorithm turns over to the second step if there is enough space. Otherwise, the frames occupied by that page are marked as free in the work-list, and the next page of the LRU-chain is examined.

2. Searching for a set of pages to be replaced (Fig. 9)

The second step is only necessary when more than one page must be replaced. The space accumulated in the work-list can be larger than the space needed, but replacement should select only a minimum number of pages and it should preserve the LRU property. Thus, actual replacement proceeds backwards through the LRU-chain. Each page is examined whether its frames correspond to the free frames detect-

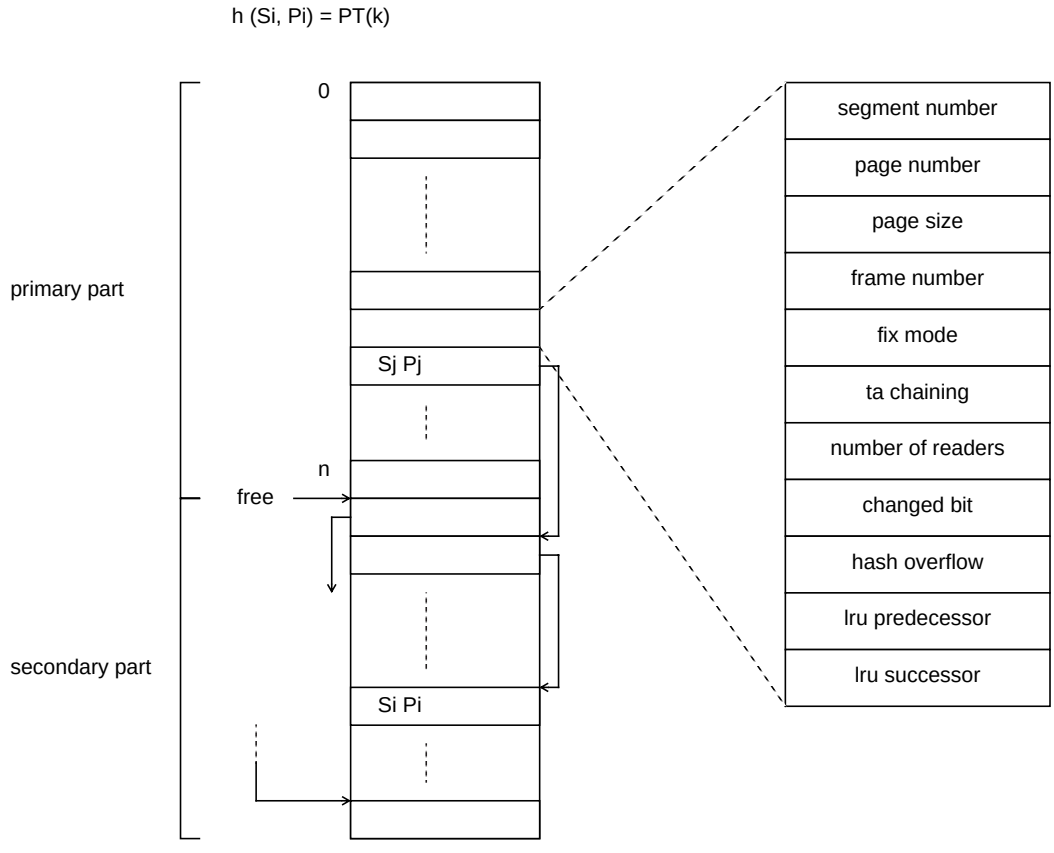


Figure 7: Page table

ed in the first step. In this case, it is checked whether the page is really necessary to build the free frames sufficient for the requested page. If not, the page is deleted from the work-list by marking the corresponding buffer frames as occupied, and the algorithm continues with the remaining number of free frames and the next page of the LRU-chain. Otherwise, the page is put on a list containing all pages to be replaced and the algorithm goes on with the next page.

This kind of procedure achieves that:

1. the number of pages to be replaced is as small as possible, and
2. the pages being replaced are as old as possible.

Fig. 9 shows an example of VAR-PAGE-LRU processing. The requested page needs 16 buffer frames. During the first step of the algorithm pages f, b, e, g, c, and d are added to the work-list in the given order (Fig. 8c). The first step stops with page d as including this page results in a number of 20 free buffer frames. During the second step the pages are examined in the other direction (d, c, g, e, b, f). Page d is always put on the list containing all pages to be replaced. Page c is removed from the work-list as the remaining number of 17 free frames is still sufficient. As a side effect page b falls out of the remaining number of examined free frames. Pages g, e, and f are put on the list, too. Page b may be dropped immediately.

The algorithm VAR-PAGE-LRU has been implemented in PL/1 and has been tested by different logical reference strings from CODASYL DBMS applications. Therefore, the different page sizes have been generated randomly. The results regarding buffer fault rate, buffer occupancy, etc. seem to be very promising [Mi86, Si88b]. Hence, it was decided to integrate the algorithm into the buffer manager.

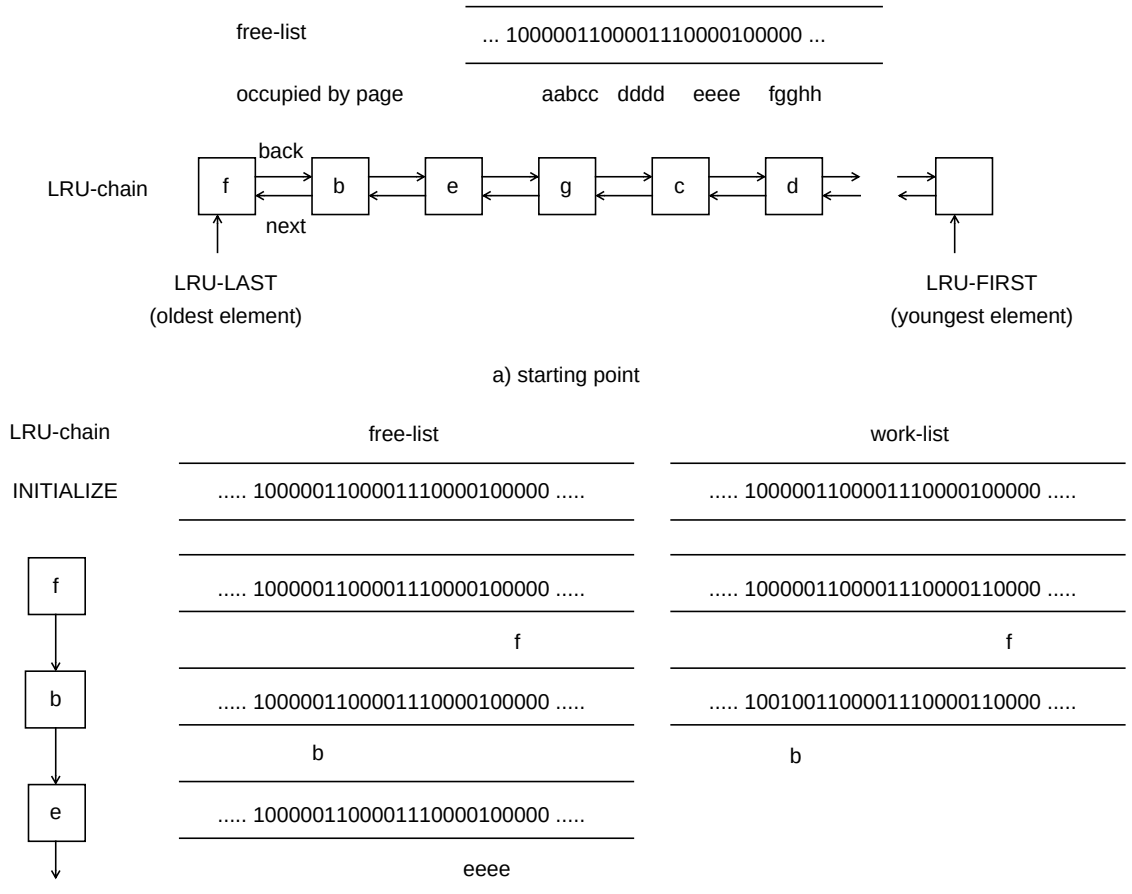


Figure 8 a and b: First step of VAR-PAGE-LRU

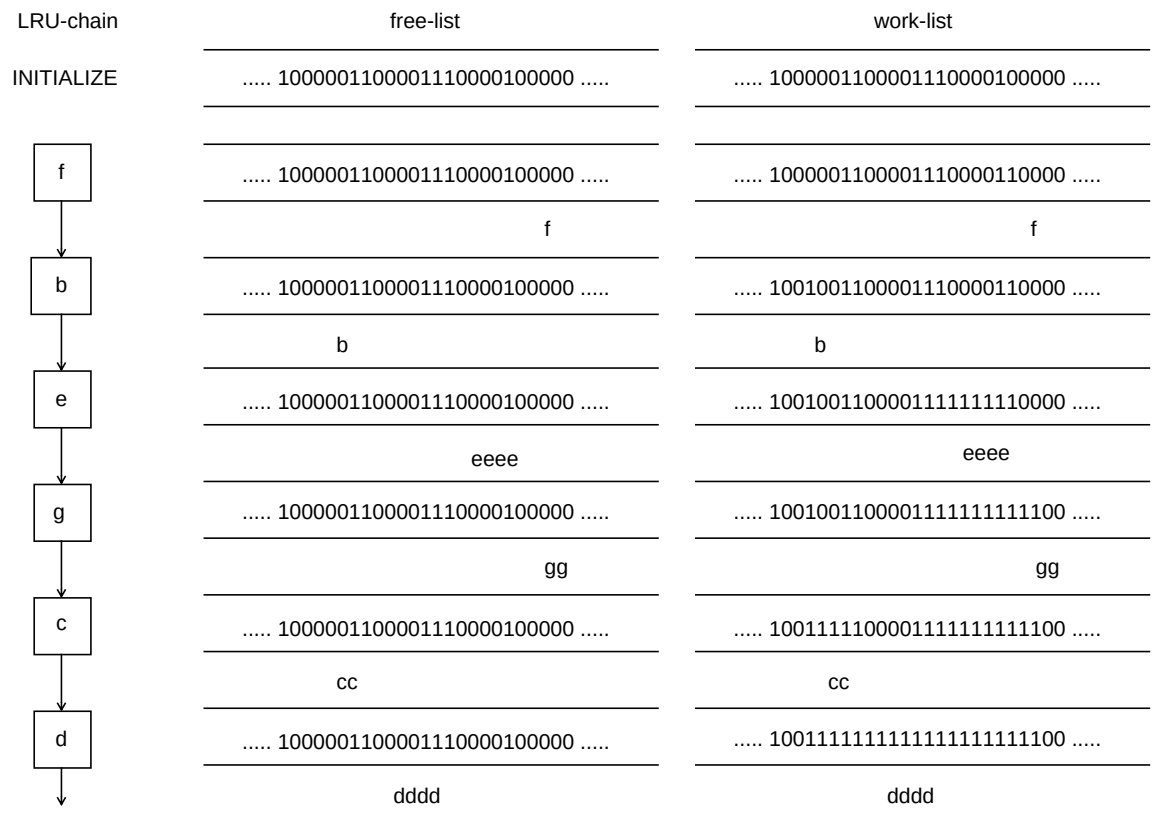
The algorithm VAR-PAGE-LRU determines all pages to be replaced. Those pages which have been modified are written back to disk, the requested page is read into the provided buffer area, and the free-list is modified according to the number of frames occupied by the page.

The actions which are necessary for a FIX operation are rather complex. The actions performed for an UNFIX operation (UNFIX-PAGE, UNFIX-PAGE-SEQUENCE, UNFIX-PAGE-SET), however, are simple. The page entry of the corresponding page (or pages) is put on the LRU-chain if there is no other transaction fixing one of these pages.

5.2 Synchronization

As already mentioned the operations at the buffer-manager interface are embedded in a transaction concept following the ACID principle [HR83]. To guarantee the third property, i.e. the isolation of transactions running in parallel, transactions have to be synchronized.

The internal structure of PRIMA is comparable to that of a DB-sharing system where some loosely coupled processors each equipped with its own database buffer share a common database on disk [HR86]. Hence, similar synchronization protocols are possible. [Ra86] gives an overview of conceivable concurrency control algorithms adapted to that environment. It can be distinguished between locking and optimistic methods as well as between centralized and distributed solutions.

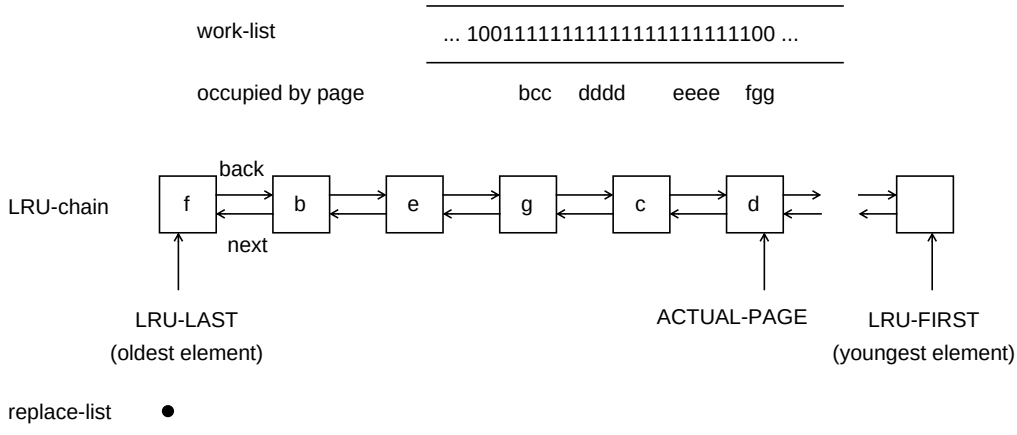


c) searching for 16 subsequent free buffer frames: turn over to step 2

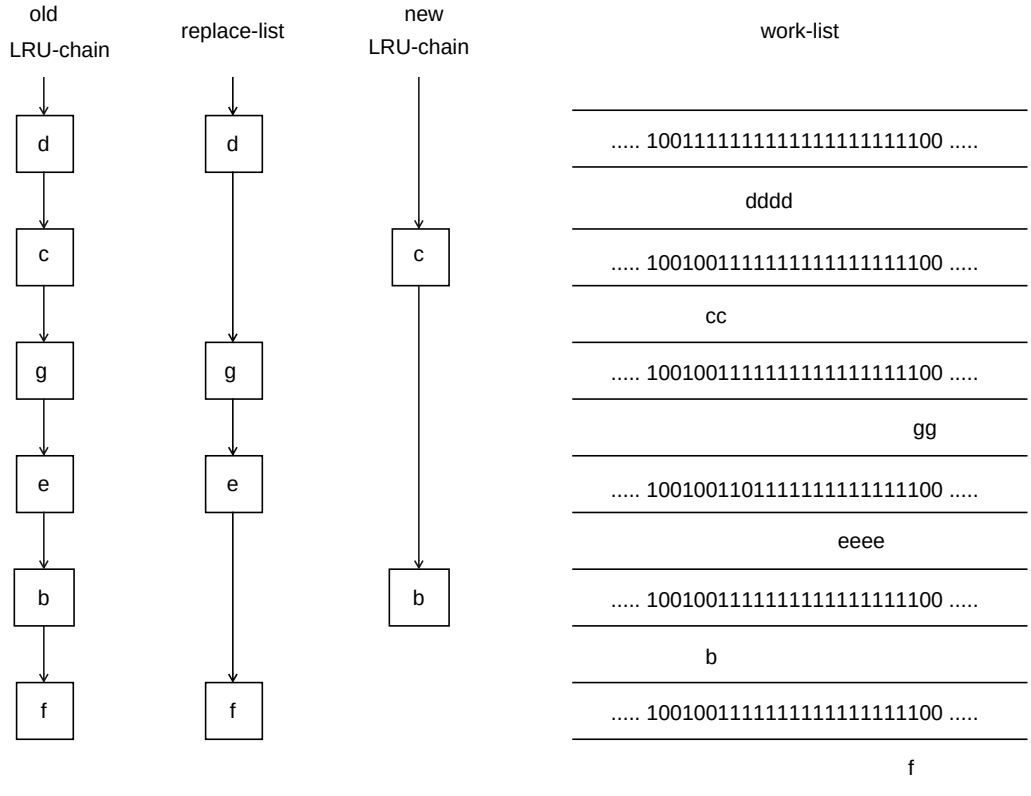
Figure 8 c: First step of VAR-PAGE-LRU

The algorithm chosen for the buffer manager of PRIMA is a locking method (RX locking) with a central lock manager [Ch87]. The communication overhead that arises when each lock request is sent to the central lock manager could be enormous. Therefore, the central lock manager is supported by local lock managers located in each processor. These local lock managers are given the authority to handle lock requests for certain pages locally. A local lock manager receives the write lock authority for a page when there is no other lock request from another local lock manager for that page. While it owns this authority it can handle write requests as well as read requests locally ("sole interest"). It loses this authority when there is a lock request (read or write) from another local lock manager. Then the authority is passed either to the central lock manager or to the other local lock manager when the corresponding page is not locked. A read lock authority is passed to each local lock manager requesting a read lock as long as there is no write lock request. While a local lock manager owns the read lock authority for a certain page it can handle all read lock requests for the page locally whereas a write lock request has to be sent to the central lock manager. When there is a write lock request, all read lock authorities are removed and the lock requests are handled by the central lock manager.

The solution of the buffer invalidation problem (see chapter 2) can be integrated into this concurrency control algorithm [CS85]. The buffer invalidation stems from the existence of a local database buffer on each processor. Multiple copies of a page may reside in various buffers at any time. Thus, modification of any copy invalidates all other copies. For the solution of this problem a so-called invalidation vector is introduced being attached to the lock control block of each page. It indicates for each of the present database buffers whether a valid or obsolete copy of the page may reside in the buffer. This information is evaluated and maintained at each lock request to the central lock manager. If a lock request can be handled locally, the present copy of that page must be valid. A problem arises if there is no copy in the buffer.



a) starting scenario (see Fig. 8c)



b) VAR-PAGELRU stops with pages f, e, g, d to be replaced

Figure 9: Second step of VAR-PAGE-LRU

In this case the corresponding page has to be requested either from the file manager, i.e. from disk, or from another buffer manager. Whether or not both ways are possible strongly depends on the method as to how modified pages are treated at end of transaction (FORCE or NOFORCE [HR83]). With a FORCE strategy all modifications of a transaction have to be written to disk at end of transaction. Hence, a valid copy of a page can always be read from disk. On the other hand, using a NOFORCE strategy, the valid copy of a page may reside in any buffer whereas the copy on disk may be obsolete. Hence, the local lock manager receives the invalidation vector (and so the information which buffer may have a valid copy of the page) together with the corresponding lock authority. Therefore, a valid copy has to be read from one of the local buffers or, when the page does not reside in any buffer, from disk.

The first action during the execution of a fix operation is a lock request to the local lock manager. The lock mode (read (R) or write (X)) corresponds to the fix mode specified in the operation. The operation and also the transaction are blocked until the lock is granted. Then the buffer manager knows whether a present copy of the page is valid or from where a valid copy has to be read.

Following the two phase commit protocol all locks of a transaction are released at end of transaction. Therefore, the complete isolation of concurrent transactions is guaranteed.

However, this complete isolation can be violated since the UNLOCK bit of an UNFIX operation can indicate that the corresponding page lock should be released immediately. In this case the calling component, i.e. the access system, has to guarantee that the page is consistent no matter whether or not the transaction is successful. That means, if the transaction is not successful the page cannot be replaced by its old contents as there may be other transactions which have already read the new contents. One possible and fruitful application of this early lock release are the parallel algorithms on access path structures, e.g. on B-trees, introduced by [Sh84].

5.3 Logging and Recovery

Logging and recovery concepts are necessary to guarantee two important transaction properties, i.e. atomicity and durability. The corresponding concepts depend on two different aspects [HR83]:

1. Which types of failure may occur ?
2. Which strategy will be used to handle modified pages ?

There are three different types of failure, but only two are considered in the buffer manager:

Transaction Failure

A transaction may not reach its normal end (commit) and has to be rolled back either on its own request (operation ABORT-TRANSACTION at the buffer-manager interface) or on behalf of the buffer manager itself (especially due to deadlocks). To guarantee atomicity all modifications of such a transaction written to the database have to be removed.

System Failure

A system failure can occur if there is a bug in the NDBS code itself or if there is a bug in the underlying operating system or hardware. In these cases all transactions which have not yet reached their normal end have to be rolled back to guarantee atomicity. On the other hand, the modifications of all transactions which have already reached their normal end must "survive".

Media Failure

A media failure occurs if some or all of the secondary storage (disks) holding the database is lost. This type of failure will not be considered here.

To cope with such failures, four different recovery actions can be distinguished:

- **transaction UNDO** (R1 recovery): to undo a single transaction during a transaction failure
- **partial REDO** (R2 recovery): to redo all transactions during a system failure which have reached their normal end
- **global UNDO** (R3 recovery): to undo all transactions during a system failure which have not yet reached their normal end
- **global REDO** (R4 recovery): during a media failure

For each of these recovery actions special log information is necessary, depending on the strategy used to handle modified pages [HR83]. The criteria characterizing this are the update propagation strategy (ATOMIC or NOTATOMIC), the page replacement strategy (STEAL or NOSTEAL), and the end of transaction processing (FORCE or NOFORCE). In a first version of the buffer manager a NOTATOMIC-STEAL-FORCE strategy should be pursued:

NOTATOMIC

Each page of a segment is related exactly to one block of the corresponding file. Writing or propagating a modified page to the database causes an update-in-place. The old image of the page on disk is lost, i.e. replaced by the modified contents.

STEAL

Modified pages of a transaction can be replaced during the execution of the transaction. Combined with a NOTATOMIC policy pages on disk can be modified at arbitrary points of time, even for incomplete transactions.

FORCE

All modified pages of a transaction have to be written back to the database at end of transaction.

In case of a FORCE policy no REDO recovery, i.e. partial REDO, is necessary, whereas in case of NOTATOMIC and STEAL UNDO recovery is needed. Therefore, the write-ahead-log principle must be used to write sufficient UNDO-information. The before-image of a page must be written to a safe place before a page (with uncommitted modifications) is propagated.

Appropriate use of logging and recovery concepts assures atomicity as well as durability (except for media failure). However, two special cases have to be considered:

1. Pages whose locks are released at unfix time have to be forced out at unfix time, and their before-images have to be deleted. Hence, R1 recovery as well as R3 recovery do not affect such pages. To take advantage of this property one has to remember this fact.
2. FPA pages are only locked as long as an FPA entry is processed. In THE case the page belonging to that FPA entry is rolled back (due to transaction abort) the FPA entry and the FPA information stored in the page may get inconsistent and UNDO recovery for the FPA page is necessary. Therefore, entry logging and recovery are used for FPA pages.

As the NOTATOMIC-STEAL-FORCE strategy was a straightforward solution for a first version of the buffer manager, other strategies are investigated with respect to log and recovery overhead and with special regard to logging and recovery concepts on higher layers, e.g. the access system. One useful consideration is the change from a NOTATOMIC policy to an ATOMIC policy based on indirect page allocation [HR83]. Examples of such mechanisms are the shadow page algorithm [Lo77] and the differential file concept [SL76]. Another possibility is the integration of the DB cache method [EB84].

6. Implementation of the Buffer-Manager Prototype

The implementation of a first version of the storage system is already finished [Bu87]. The implementation language is LADY (Language for Distributed Systems [WM85]). LADY distinguishes among three language layers:

- A system consists of a set of teams interconnected via ports. The interconnection structure is given by either directed port-to-port channels or by logical busses (multicast, broadcast).
- Each team (more precisely: each team type) is decomposed into a set of processes and monitors. Monitors serve for intra-team communication.
- At the module level the algorithmic behavior of each module type (process, monitor, class, procedure) is specified. Modules can be compiled separately.

At the system level the complete NDBS prototype, i.e. PRIMA, the application layer, and the application programs, consists of a set of teams belonging to different team types (Fig. 10). There is one team type realizing the application layer and the application programs. The next type embodies PRIMA (except the file manager) and the third type only consists of the file manager. Furthermore, there is yet another team type for the central lock manager. The number of teams of each team type depends on the underlying hardware architecture (Fig. 2). One file manager and one central lock manager will run together on the processor with direct connections to the external devices. One processor is reserved for the application, and all other processors will run exactly one PRIMA.

At the team level the team type PRIMA is decomposed in a large number of processes and monitors. Although LADY does not support subteams within a team type, it is useful to introduce subteams at the description level (Fig. 11). Namely, a subteam consists of a set of processes and monitors and represents an isolated unit within a team. Within PRIMA four subteams can be distinguished: one subteam for the data system, one for the access system, one for the buffer manager, and lastly one for the meta-data system. The meta-data subteam manages the meta-data of all other PRIMA components [We87]. Subteams are able to communicate with each other via monitors (in a future extension of LADY [WHB86] via monitors and areas; areas are shared memory for some processes). Therefore, each subteam offers op-

erations at its interface as entries of one or more monitors and uses monitor entries to call services of other subteams. Fig. 11 shows a diagrammatic view of the internal structure of the team type PRIMA.

The buffer-manager subteam offers two monitors (the buffer-manager interface and the buffer itself), whereas it needs one monitor for the communication with the meta-data subteam (meta-data interface) (Fig. 12). The buffer monitor comprises the database buffer since for the moment this is the only way to share memory or data. After an extension of LADY the monitor will be replaced by an area as shared memory. The buffer-manager-interface monitor represents the proper interface of the buffer-manager subteam. Each operation described in chapter 4 as well as the operations BEGIN-OF-TRANSACTION, END-OF-TRANSACTION, and ABORT-TRANSACTION correspond to one monitor entry. The actions performed for each entry are always the same: A request element is inserted into a corresponding request queue and the control is returned to the calling subteam. To get an answer back, the calling process has to use the monitor entry GET-ANSWER. This monitor entry returns an answer element for one specific request. If there is no answer for the corresponding request, the calling subteam is blocked until a corresponding answer element is inserted into the answer queue. For that purpose, the monitor entry PUT-ANSWER is used by the buffer-manager subteam, whereas GET-COMMAND is used by the buffer-manager subteam to get a request. Hence, the buffer-manager subteam and the calling subteam can work in an asynchronous way. For that reason, all monitors forming an interface are organized in the same way.

To process a request the buffer-manager subteam is divided into a set of process and monitor types (Fig. 12):

- All operations concerning segments (see section 4.1) are handled by the FPA manager. Therefore, the FPA manager uses special operations offered by the buffer manager to create, expand or delete a segment and to fix or unfix an FPA page. These operations are not available outside the buffer-manager subteam.
- The buffer manager handles all operations concerning pages and page sets (see section 4.2 and 4.3). Furthermore, it uses the FPA manager to determine the page type of a requested page, the lock manager to request a corresponding lock, the log manager to write the corresponding log information, and the recovery manager to initialize transaction abort. The buffer manager communicates with the file manager via different input and output ports. These ports are maintained by additional send and receive processes.
- The lock manager [Kr88] corresponds to the local lock manager described in section 5.2. It handles all lock requests either locally or by requesting the central lock manager via input/output ports.
- The log manager [Ho88] handles all log information during normal transaction processing.
- The recovery manager [Ho88] performs partial as well as global UNDO, i.e. roll-back processing of a single transaction after transaction abort and roll-back processing of all incomplete transactions after a system failure.
- Sender and receiver processes are required since send and receive operations offered by the underlying operating system DISTOS are synchronous and all input and output ports are typed. Therefore, it is necessary to associate a receive process with each input port to allow for parallelism of PRIMA and the file manager.
- The lock-manager-interface monitor, etc. are used for communication purposes between the different processes. They are organized similar to the buffer-manager-interface monitor.

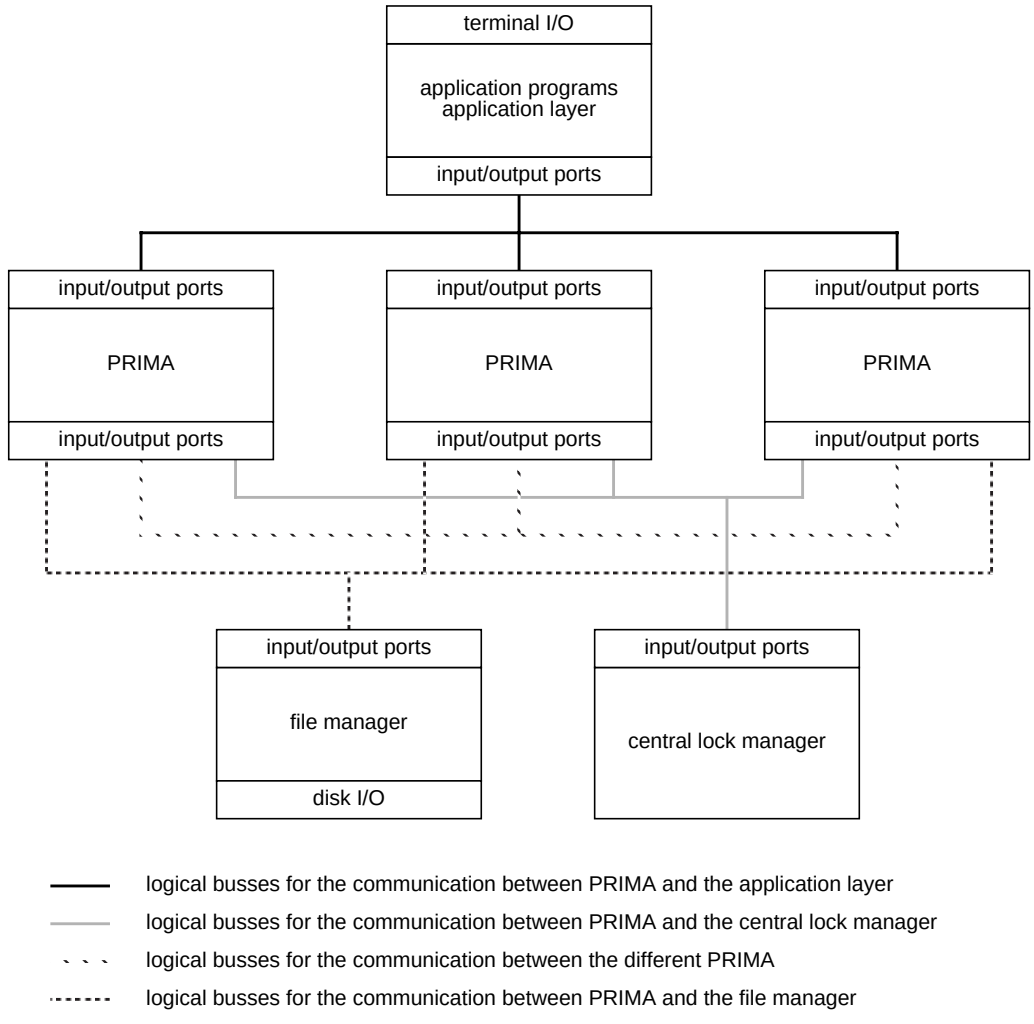


Figure 10: Possible team network of the complete NDBS prototype

- Tne exception is the global-data monitor which includes all global data structures of the buffer manager (e.g. page table etc.). Therefore, several buffer managers may run in parallel sharing data by the global-data monitor.

This design of the PRIMA team and the buffer-manager subteam supports parallelism at two different levels:

- Within PRIMA the different subteams can run in parallel. While one subteam performs an operation, the other subteam requesting this operation can continue its work until it needs the answer for the operation.
- Within the buffer-manager subteam more operational power at the buffer-manager interface can be achieved depending on the number of buffer-manager and FPA-manager processes.

The presented design considerations allow for a substantial degree of freedom. Proper values - like useful number of processes, degree of parallelism, etc. - still have to be examined in more detail.

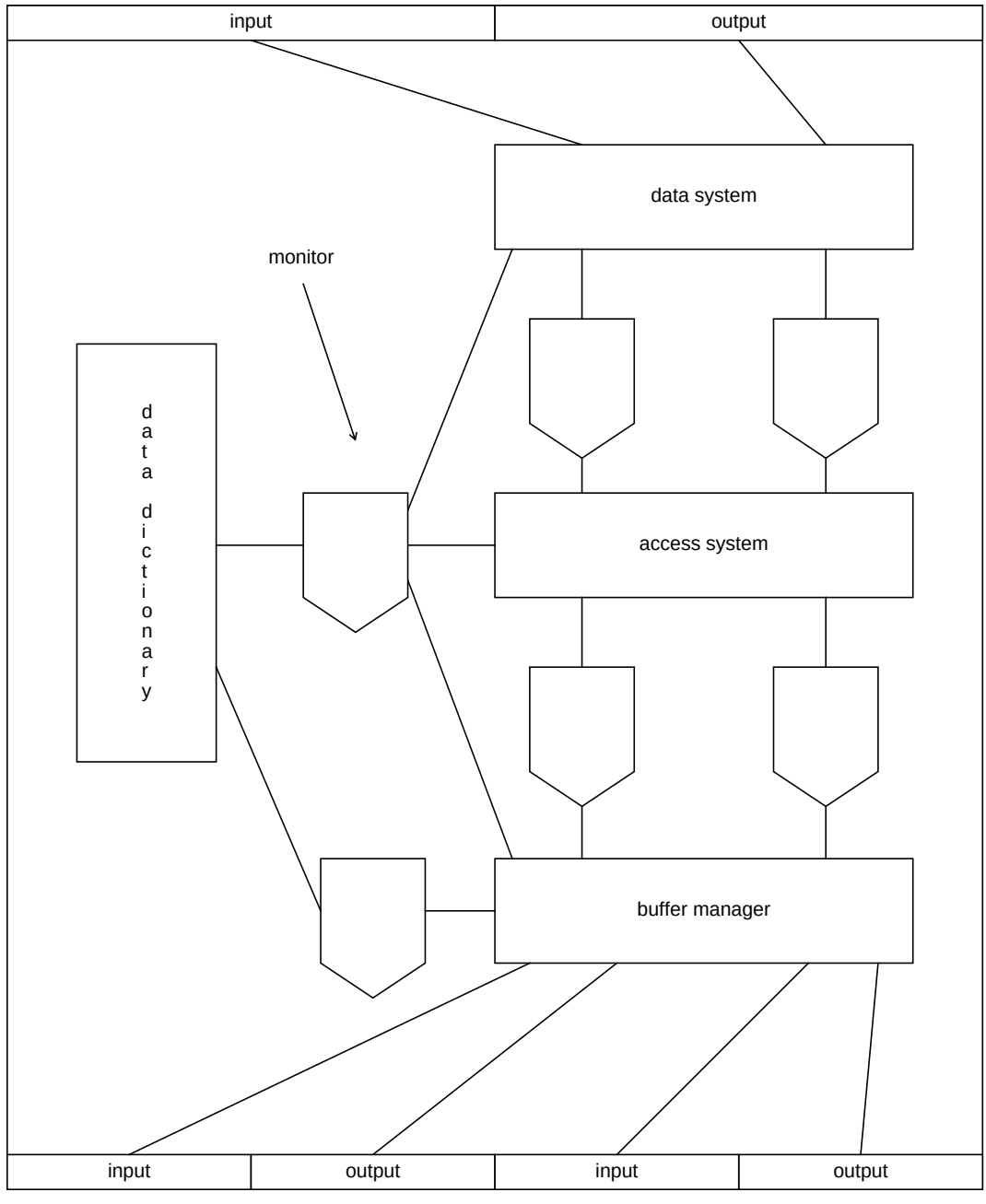


Figure 11: Internal structure of the team type PRIMA

7. Conclusions

Non-standard applications require a sophisticated data management component to be considered during the design of a non-standard database system. The main demands on the storage system of such an NDBS concern support of

- bjects of different size, especially very long objects,
- varying number of objects, and
- in particular physical clustering.

Therefore, new objects and operations at the storage-system interface are necessary. In PRIMA, a prototype implementation of the molecule atom data model, these are

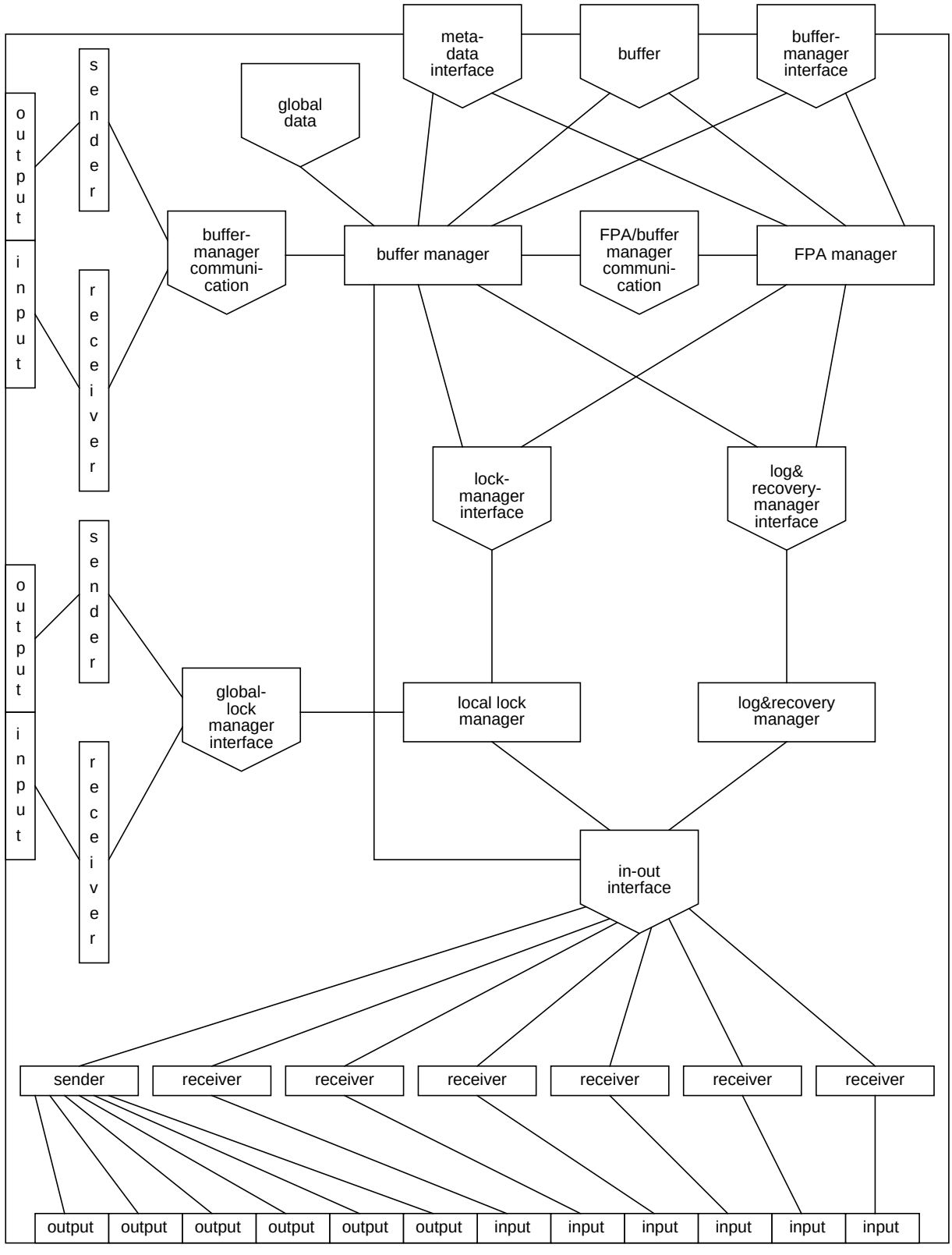


Figure 12: Structuring of the buffer-manager subteam

- segments of different page sizes,
- operations to expand and shrink a segment,

- page sequences and corresponding operations to handle a predefined page set, and
- operations to handle an arbitrary set of pages.

Furthermore, the administration of free place in a segment and the pages of a segment is integrated into the storage system. Hence, the storage-system interface of PRIMA is expanded by operations performing free place administration.

All operations at the storage-system interface are embedded in a transaction concept following the ACID principle. A log and recovery manager is integrated into the storage system to guarantee atomicity and durability, and a lock manager is integrated to guarantee isolation by synchronizing all page requests. This lock manager is also used to solve the buffer invalidation problem that depends on the underlying hardware architecture, a set of loosely coupled processors with local memory.

The implementation of a first version of the storage system of PRIMA is already finished. The implementation language is LADY and the underlying operating system is DISTOS. LADY as well as DISTOS are designed to support the implementation of distributed systems. To test this first version of the storage system, real reference strings as well as generated reference strings will be used [Se88].

Future work regarding PRIMA will concentrate on the access system and the data system. This will test and verify the design and implementation of the first version of the storage system. Modifications will finally lead to a new version.

Independently, the log and recovery concept (e.g. entry logging) used within the storage system has to be reconsidered with respect to the integration into the overall log and recovery concept of PRIMA. Furthermore, atomic update propagation strategies can be investigated and implemented without influencing the design and implementation of the higher levels.

References

As76 Astrahan, M.M., et al.: SYSTEM R: A Relational Approach to Database Management, in: ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.

As81 Astrahan, M.M., et al.: A History and Evaluation of SYSTEM R, in: Communications of the ACM, Vol. 24, No. 10, October 1981, pp. 632-646.

BB84 Batory, D.S., Buchman, A.P.: Molecular Objects, Abstract Data Types and Data Models: A Framework, in: Proceedings of the 10th International Conference on Very Large Databases, Singapore, 1984, pp. 172-184.

Bu87 Bub, E.: Entwurf und Implementierung der Systempufferverwaltung von PRIMA, University Kaiserslautern, 1987.

Ch85 Chou, H.T., et al.: Design and Implementation of the Wisconsin Storage System, in: Software - Practice and Experience, Vol. 15, No. 10, October 1985, pp. 934-962.

Ch87 Christmann, H.-P.: Ein Algorithmus zur Systempufferverwaltung und Synchronisation in einem lose gekoppelten Mehrrechner-Datenbanksystem, in: Proceedings of the GI/NTG Conference on Communication in Distributed Systems, Aachen, ed.: N. Gerner, O. Spaniol, Informatik-Fachberichte No. 130, Springer, Berlin Heidelberg New York Tokyo, 1987, pp. 412-425.

CHMS87 Christmann, H.-P., Härder, T., Meyer-Wegener, K., Sikeler, A.: Which Kinds of OS Mechanisms Should Be Provided for Database Management?, in: Proceedings of the International

Workshop on Experiences in Distributed Systems, Kaiserslautern, Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York Tokyo, 1987.

CS85 Christmann, H.-P., Schöning, H.: Simulation von Algorithmen zur Systempufferverwaltung und Synchronisation in einem lose gekoppelten Mehrrechner-Datenbanksystem, SFB 124 Research Report No. 21/85, University Kaiserslautern, 1985.

DPS86 Deppisch, U., Paul, H.-B., Schek, H.-J.: A Storage System for Complex Objects, in: Proceedings of the International Workshop on Object Oriented Database Systems, Asilomar, ed.: K. Dittrich, U. Dayal, 1986, pp. 183-195.

Da86 Dadam, P., et al.: A DBMS Prototype to Support Extended NF2-Relations: An Integrated View on Flat Tables and Hierarchies, in: Proceedings of the ACM SIGMOD Conference, Washington D.C., 1986, pp. 356-367.

EB84 Elhard, K., Bayer, R.: A Database Cache for High Performance and Faster Restart in Database Systems, in: ACM Transactions on Database Systems, Vol. 9, No. 4, 1984, pp. 503-525.

EH84 Effelsberg, W., Härder, T.: Principles of Database Buffer Management, in: ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984, pp. 560-595.

Fr87 Franz, B.: Konzeption und Implementierung eines Dateisystems für das DISTOS-Betriebssystem, University Kaiserslautern, 1987.

Hä78 Härder, T.: Implementierung von Datenbanksystemen, Carl Hanser, München, 1978.

Hä87 Härder, T., et al.: PRIMA - a DBMS Prototype Supporting Engineering Applications, in: Proceedings of the 13th International Conference on Very Large Data Bases, Brighton, 1987, pp. 433-442.

Ho88 Hoffmann, C.: Implementierung der Log- und Recovery-Komponente auf Seitenebene, University Kaiserslautern, 1988.

HR83 Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys, Vol. 15, No. 4, 1983, pp. 287-317.

HR85 Härder, T., Reuter, A.: Architektur von Datenbanksystemen für Non-Standard-Anwendungen, in: Proceedings of the GI Conference on Database Systems for Office, Engineering, and Science Environments, Karlsruhe, ed.: A. Blaser, P. Pistor, Informatik-Fachberichte No. 94, Springer, Berlin Heidelberg New York Tokyo, 1985, pp. 253-286.

HR86 Härder, T., Rahm, E.: Mehrrechner-Datenbanksysteme für Transaktionssysteme hoher Leistungsfähigkeit, in: Informationstechnik, Vol. 28, No. 4, 1986, pp. 214-225.

Kr88 Kreiselmeier, J.: Implementierung der Sperr-Komponente auf Seitenebene, University Kaiserslautern, 1988.

Kü85 Küspert, K.: Fehlererkennung und Fehlerbehandlung in Speicherungsstrukturen von Datenbanksystemen, Informatik-Fachberichte No. 99, Springer, Berlin Heidelberg New York Tokyo, 1985.

Lo77 Lorie, R.: Physical Integrity in a Large Segmented Database, in: ACM Transactions on Database Systems, Vol. 2, No. 1, 1977, pp. 91-104.

Lo84 Lorie, R., et al.: Supporting Complex Objects in a Relational System for Engineering Databases, in: Query Processing in Database Systems, ed.: Kim, W., Reiner, D.S., Batory, D.S., Springer, Berlin Heidelberg New York Tokyo, 1984, S. 145-155.

Mi85 Mitschang, B.: Charakteristiken des Komplex-Objekt-Begriffs und Ansätze zu dessen Realisierung, in: Proceedings of the GI Conference on Database Systems in Office, Engineering,

and Science Environments, Karlsruhe, ed.: A. Blaser, P. Pistor, Informatik-Fachberichte No. 94, Springer, Berlin Heidelberg New York Tokyo, 1985, pp. 382-400.

Mi88 Mitschang, B.: The Molecule-Atom Data Model, in: The PRIMA Project - Design and Implementation of a Non-Standard Database System, T. Härder (ed.), SFB 124 ResearchReport No. 26/88, University Kaiserslautern, 1988.

Mi86 Michels, M.: Implementierung und Test einer LRU-Ersetzungsstrategie für Seiten unterschiedlicher Größe, University Kaiserslautern, 1986.

Ne87 Nehmer, J., et al.: The Multicomputer-Project INCAS - Objectives and Basic Concepts, in: IEEE Transactions on Software Engineering, Vol. SE-13, No. 8, 1987, pp. 913-923, 1987.

Pa84 Paul, H.-B., at al.: Überlegungen zur Architektur eines "Non-Standard"-Datenbankkernsystems, Research Report DVSI-1984-A2, Technical University Darmstadt, 1984.

Pa87 Paul, H.-B., et al.: Architecture and Implementation of the Darmstadt Database Kernel System, in: Proceedings of the ACM SIGMOD Conference, San Fransisco, 1987, pp. 196-207.

Ra86 Rahm, E.: Buffer Invalidation Problem in DB-Sharing Systems, Research Report No. 154/86, University Kaiserslautern, 1986.

Sch88 Schöning, H.: The PRIMA Data System: Query Processing of Molecules, in: The PRIMA Project - Design and Implementation of a Non-Standard Database System, T. Härder (ed.), SFB 124 Research Report No. 26/88, University Kaiserslautern, 1988.

Se88 Sellentin, U.: String-Manager und String-Generator zum Test der Systempufferverwaltung, University Kaiserslautern, 1988.

Sh84 Shasha, D.E.: How to Design a Concurrent Algorithm for Your Favorite Search Structure, TR-13-84, Harvard University, 1984.

Si80 Sidle, T.W.: Weakness of Commercial Database Management Systems in Engineering Applications, in: Proceedings of the 17th Design Automation Conference, Minneapolis, 1980, pp. 57-61.

Si88a Sikeler, A.: The Key Concepts of the PRIMA Access System, in: The PRIMA Project - Design and Implementation of a Non-Standard Database System, T. Härder (ed.), SFB 124 ResearchReport No. 26/88, University Kaiserslautern, 1988.

Si88b Sikeler, A.: VAR-PAGE-LRU: A Buffer Replacement Algorithm Supporting Different Page Sizes, in: Proceedings of the Extending Data Base Technology Conference, Venice, Lecture Notes in Computer Science No. 303, Springer, Berlin Heidelberg New York Tokyo, March 1988, pp. 336-351.

SL76 Severance, D.G., Lohmann, G.M.: Differential Files: Their Application to the Maintenance of Large Databases, in: ACM Transactions on Database Systems, Vol. 1, No. 3, 1976, pp. 256-267.

SS86 Schek, H.-J., Scholl, M.H.: The Relational Model with Relation-Valued Attributes, in: Information Systems, Vol. 2, No. 2, 1986, pp. 340-355.

St81 Stonebraker, M.: Operating System Support for Database Management, in: Communications of the ACM, Vol. 24, No. 7, 1981, pp. 412-418.

UDS UDS - Universelles Datenbanksystem, User Manuals Version 5.0, Siemens AG, München, 1986.

We87 Weber, B.: Implementierung eines einfachen Verwaltungssystems für die Metadaten in PRIMA, University Kaiserslautern, 1987.

WHB86 Wybranietz, D., Haban, D., Buhler, P.: Some Extensions of the LADY Language, SFB 124 Research Report No. 26/86, University Kaiserslautern, 1986.

WM85 Wybranietz, D., Massar, R.: An Overview of LADY - A Language for the Implementation of Distributed Operating Systems, SFB 124 Research Report No. 11/85, University Kaiserslautern, 1985.

Appendix: Operations offered by the storage system

INIT_SEGMENT	IN:	transaction identifier
		segment name
		page size
		segment type
		first allocation
	OUT:	segment_number
		returncode
DELETE_SEGMENT	IN:	transaction identifier
		segment number
	OUT:	returncode
GET_FREE_PLACE	IN:	transaction identifier
		segment number
		number of bytes
	OUT:	list of page numbers
		returncode
MODIFY_FPA	IN:	transaction identifier
		segment number
		page number
		free place
	OUT:	returncode
INCREMENT_FPA	IN:	transaction identifier
		segment number
		page number
		increment
	OUT:	returncode
DECREMENT_FPA	IN:	transaction identifier
		segment number
		page number
		decrement
	OUT:	returncode
FIX_PAGE	IN:	transaction identifier
		segment number
		page number
		fix mode

	OUT:	page address returncode
UNFIX_PAGE	IN:	transaction identifier segment number page number changed unlock
	OUT:	returncode
FIX_PAGE_SEQUENCE	IN:	transaction identifier segment number page-sequence number fix mode
	OUT:	list of page addresses returncode
UNFIX_PAGE_SEQUENCE	IN:	transaction identifier segment number page-sequence number changed unlock
	OUT:	returncode
FIX_PAGE_SET	IN:	transaction identifier list of (segment number and page number) and/or (segment number and page-sequence number) fix mode
	OUT:	list of page addresses returncode
UNFIX_PAGE_SET	IN:	transaction identifier list of (segment number and page number) and/or (segment number and page-sequence number) changed unlock
	OUT:	returncode
PAGE_CONTEST	IN:	transaction identifier list of (segment number and page number) and/or (segment number and page-sequence number) fix mode
	OUT:	segment number page number or page-sequence number list of page addresses returncode