A Storage Manager for Semistructured Databases

Amparo López Gaona[1] and Egar Arturo García Cárdenas[2]

[1] Facultad de Ciencias, Universidad Nacional Autonóma de México, Mxico D.F. 04510

[2] Dirección General de Bibliotecas, Universidad Nacional Autonóma de México, Mxico D.F. 04510

Abstract. Semistructured data and their databases have emerged as an attempt for managing flexible information, which is in constant growing and its needs are difficult to predict. A semistructured database manager system needs to store semistructured data in a persistent way into secondary storage devices, the performance and proficiency aspects must be considered.

In This work we present a native semistructured database storage manager and the main aspects of its design and functionality. The storage manager includes mechanisms for grouping multiple devices, intermediate memory management, free space control and the internal representation of semistructured data. For implementing the internal representation of semistructured data and their functionality the use of many data structures is required. The design of the semistructured database storage was completed with its implementation. The aim of this work is to contribute in the construction of a semistructured database manager system.

1 Introduction

Semistructured databases have emerged in the last years as an alternative to overcome the difficulties in other database paradigms like the relational and object oriented models, these difficulties are associated with too flexible information (i.e. the semistructured data don't follow to the defined formats), in constant updating, which needs are difficult to predict and is scattered in several sources.

For using a semistructured database is necessary to have a management system, particularly the semistructured database manager system (SSDBMS) must store data in a persistent way. We present the constructions built to store semistructured databases in secondary storage devices. This storage involves several subproblems like the use of intermediate memory, the grouping of multiple devices, the use of raw and cooked partitions, the management of free space, the way to represent semistructured data in a low level and the incorporation of large objects (BLOBs and CLOBs).

2 The Primitive Storage

Operating systems implements their own minimum reading-writing units called *blocks*, a block can be formed for a single or several sectors, depending on the file system configuration, generally the operating systems have the option for changing the blocks' size depending on their needs, for using a computer system as a database server a big block's size is frequently used.

For store manager, database manager systems have two options: To use the storage system that the operating system provides, through the file system, or to implements its own storage system. When the database manager system implements its own storage system, it also defines a minimum reading-writing unit, in this context this unit is called *page*, a page can be formed for one or many blocks.

In the database manager systems page's size is given according with the storage's space needed, for large information's quantities large page's sizes are used and for small information's quantities small page's sizes are used, the objective is to minimize the number of disk's accesses. A large page's size can increment the reading-writing time to the secondary storage device cause the block's size depends of the operating system, but with a good organization of information and free space the number of disk's accesses can be reduced.

2.1 Intermediate Memory Management

To increment the performance of the storage, the computational systems use *buffers*, in this work buffers are used to storage a specified quantity of pages. To attend the input-output requests processes must verify that the requested page is kept in the buffer, if it doesn't they have to access the disk to retrieve the page. The buffer's size is limited then all pages can't be kept in it. To chose the pages to hold in the buffer and the pages to replace, the replace algorithms are used [5, 4] the objective of these algorithms is to minimize the number of page's replaces which causes a reduction of disk's access, a disk access is approximately 10000 times slower than a memory access, an adequate replace algorithm increments the storage's performance.

This work uses a data structure compound of a hashing table and a double linked circular list (fig. 1), the hashing table is used to find the pages in the buffer according within its address, the circular list is used to implement the replace algorithms (LRU, MRU, FIFO, LIFO or CLOCK). For each page two bites are kept: one for reference and one for writing, the writing bit turns on only when the page's content changes, in that way when a page is replaced from the buffer it is updated in disk only if its writing bit is on, it reduces the disk's accesses.

2.2 Use of multiple devices

Sometimes the size of a database is so big that is not possible to store it in only one device, some database manager systems have mechanisms to group many

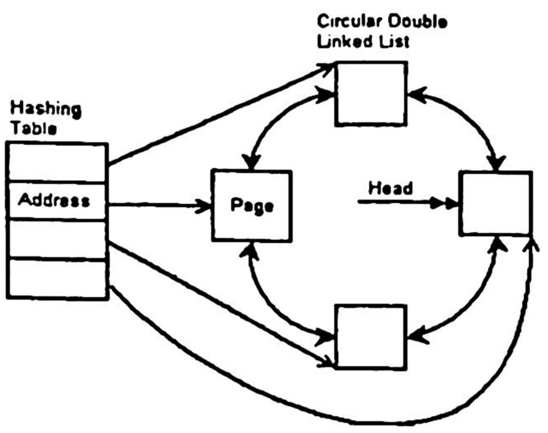

Fig. 1. Buffer implementation.

devices into one database. Usually storage devices are one partition of some device, not the device itself. There are two kinds of partitions: raw and cooked. The DBMS controls directly raw partitions without use the file system of the given operating system and formats them. The cooked partitions are mounted on top of the file system and the operating system formats them.

In terms of efficiency, the best option for the DBMS is to use raw partitions, this avoids the use of operating system's file operations, which allows the DBMS to implement its own specialized structures to achieve the desired efficiency levels. The DBMS has several options for the use of raw and cooked partitions,in the extremes we have: To use the file system of the operating system, or to use and format the partitions as it needs them. There are some intermediate options, for example to use a big file (given by the file system), in this file the DBMS creates its own format. There is consensus on the use of raw partitions to achieve the better levels of efficiency, however, some operating systems, like Windows, don't allow the use of raw partitions; other systems, like UNIX, allow the use of this partitions and it is possible to use them as if they were other devices, so it is possible to mount them over the file system and use them as random access files.

Our solution to the problem of grouping several storage devices, raw or cooked, it's to use a technique based in *chunks*, as is done in INFORMIX[7]. A chunk is defined as a file o as part of a file. There exists two kinds of chunks: *flexible chunks* and *fixed chunks*. Flexible chunks are used under the file system, and are restricted to the limits given by the operating system, they are usually implemented on cooked partitions, in this way the operating system is responsible of the storage management; the described technique is a good option for small databases. Fixed chunks need a precise storage management, but in systems like UNIX they allow the incorporation of raw partitions.

The SSDBMS works with the chunks as if they were different storage devices, in this way it is possible to group spaces even in different disks or partitions. Chunks are seen as a sequence of pages, so it is easy to group the devices, it is just a matter of defining the group of chunks to use. To access a particular page is neccessary to give the chunk number and the page number within this chunk.

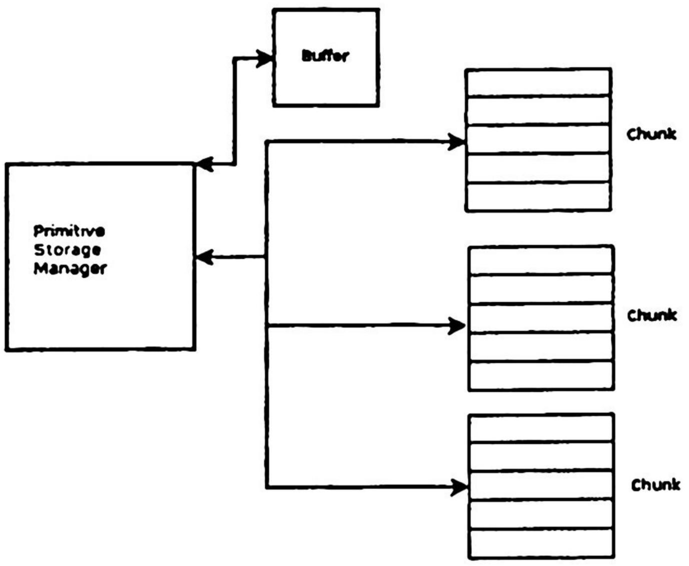

Fig. 2. Primitive storage.

3 Advanced storage

SSDMS requires variable size data structures to store its data, this is due to the nature of semistructured data. Usually, the techniques used to store variable size data structures consist in a distribution of the data in fixed size boxes, in some storage medium; the idea is to ease the access as well as the management of free space, these techniques also avoid the external fragmentation.

We have designed our storage model for the SSDBMS based on these techniques; the model is called the *advanced storage*. This model is based on a three levels organization: *Segment*, *space* and *unit*. The segment is an array of spaces, each space is an array of units, and each unit corresponds to the fixed size boxes where the information will be distributed, so the units can be free or used (fig. 3). All the units have the same size within each segment, although units in different segments can be of different sizes. The size of the unit is calculated as the DBMS requires storage for some data. To access one specific unit it is necessary to specify: the segment number, the space number in the segment and the unit number in the space. In this storage level we give mechanism to read or write in a unit, to find a free unit, and to free or use one unit.

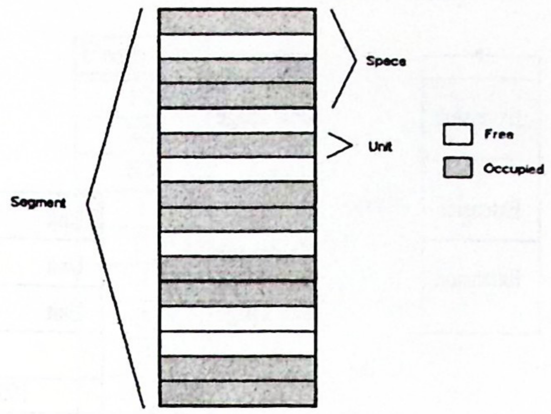

Fig. 3. Advanced storage model.

3.1 Implementation of the advanced storage

To implement our model we use the primitive storage, in this way the spaces correspond to chunks and the units fit in the pages, the constrain is that the unit size should be smaller than the page size.

Spaces are associated with chunks, internally each space has three organization levels: *area*, *extension* and *page of units*. The areas and extensions were created with the purpose of implement the management of the free space, this is often done with the use of *bitmaps*, so the space is composed of areas, each area is composed of extensions and each extension is composed of pages of units. The page can be a page of units, a bitmap or a descriptor. The first page of the space is a descriptor which has the number of the last unit used, this number is used to calculate the part of space that has been used, in this way we don't need to fully format the space, we format the space as it is used; this is an advantage when the full size of the storage is unknown, that is, when the flexible chunks are used.

Each space has one descriptor and several areas, one after the other. The first page of one area is a bitmap that indicates which extensions are free and which are used, the position of one bit in the map correspond with the position of the extension in the area, a zero bit means the extension is free, otherwise the extension is full. Just after this bitmap are the extensions, one after the other (fig. 4).

The first page of an extension is a bitmap that indicates which pages of units are available and which are not, again a zero number means that the page of units is available. This is done in the same way the position of bit in the map correspond with the position of the page of units in the extension. After this bitmap we find the pages of units, one followed by the other one (fig. 4).

The pages of units can have one or more units, if the size of the unit has enough space for several pages then there is one bitmap at the end of the pages of units, if it only has space for one unit then we don't have the bitmap (fig. 4).

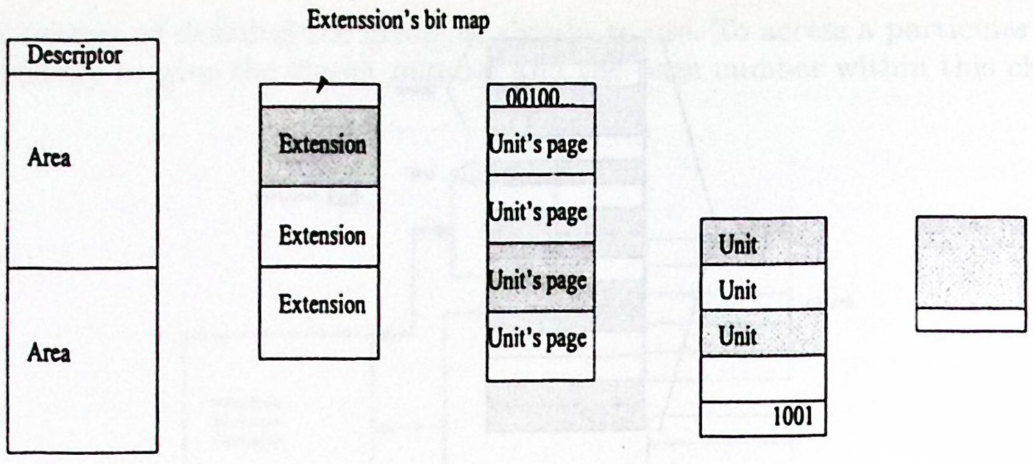

Fig. 4. Distribution of one space, one area, one extension and pages of units.

The bitmap in the page of units marks if the units are free or used, the position in the bitmap is the position of the unit within the page of units.

The size of the extensions and areas depends on the page size, if this size is P (in bytes), the size of one extension is $(8P + 1)P$, because it is possible to store the status of $8P$ pages in one page and it requires another page for the bitmap. Similarly the size of one area is $(8P(8P+1)+1)P$, because one area can store the status of $8P$ extensions and each extension has $8P + 1$ pages. To find one free unit in one area is necessary to access its bitmap and find one available extension, then we can go to the bitmap of the extension to search the available page of units. If we relate each page access to one access to the disk then we need three disk operations to locate a free unit, when we don't have free units it only takes one disk operation. Table 1 shows some page sizes and the resulting area size, as we can see, with a small number of disk operations we can organize the free space in a big amount of space, for example, with a $4K$ page we need just three disk operations to find a free unit in 4 terabytes of information.

Once we know the unit number in a space, we only need one disk operation to get it, because we can calculate the page based on the area and extension number. We may not achieve this results when we use cooked partitions, this is because we have to take into account the operations needed by the operating system to get to the desired page. When we use raw partitions we should consider the sector size; however, this is a good way to measure the performance of the storage model.

The page addresses need 8 bytes within the space, as this is the biggest address space for a programming language; the space addresses need 4 bytes, this is due to the limits in the programming language for the addressing of chunk tables in memory. So we need 12 bytes to find a page in a segment. This address size is almost unlimited for the space size of the current systems.

Table 1. Pages and areas sizes.

Page size	Area size
128 B	128.125122070 MB
256 B	1.000488520 GB
512 B	8.001953602 GB
1 KB	64.007813454 GB
2 KB	512.031251907 GB
4 KB	4.000122074 TB
8 KB	32.000488289 TB
16 KB	256.001953140 TB
32 KB	2.000007629 PB
64 KB	16.000030518 PB
128 KB	128.000122070 PB
256 KB	1.000000477 HB
512 KB	8.000001907 HB
1 MB	64.000007629 HB
2 MB	512.000030518 HB
4 MB	4.000000119 ZB

4 Storage of a Semi Structured Data Base Management System

We need to use five segments of the advanced storage for the main components of a SSDBMS; the five segments are: *ID* to store the data descriptors; *CONTENT* the place used to store the contents of the semistructured data which make the database; *LOB* it is used to store the contents of the large objects (LOBs); *PARENTS* used to store the collections of parents for the semistructured data; and *DICTIONARY* used to store the data dictionary, which contains the name for the ssd-tables and their roots.

4.1 The ID segment

In the ID segment we store descriptors for the semi structured data which make the data base, this semi structured data can be primitive or non primitive; primitive data may be simple or large objects (LOB's). We can say that primitive data can be handled in memory and the large objects can not. Non primitive data are collections or sets of other labeled semi structured data.

For the descriptors we need to store the data type, the amount of the ssd-tables for which this data is a root (this is the quantity of references from the dictionary towards the data), the pointer to its collection of parents and the pointer to its content.

As most of the semi structured data have only one parent, we may reserve one slot in the descriptor to store the identifier of this parent, which may save space in the PARENTS segment with their respective savings in the disk operations.

Also, most semi structured data are of some primitive type, so we can reserve other slot in the descriptor to store the contents of some primitive data, which lead to more savings in the CONTENT segment and their respective savings in disk accesses.

Each semi structured data has a unique identifier, we can have faster findings for the semi structured data if this identifier corresponds to its physical position, as can be its descriptor. In the ID segment we store the descriptors for the semi structured data, each unit represents one descriptor, the space number and the unit number within the space make the identifier for this data.

We have three types of semi structured data, so we have three types of descriptors; one data descriptor uses 96 bytes, so the size of the unit in the segment is 96 bytes.

For the three types of descriptors we should store: the number of references from the dictionary, the identifier of the first parent, the address of the root of the tree used for the collection of parents in the PARENTS segment, the address of the list head used for the parents collection in the PARENTS segment and the data type.

For the descriptor of simple primitive data we need to store: the address to one unit in the CONTENT segment which represents an extension when the data content is bigger than the descriptor, the data size and the first 32 bytes of the data content.

For the descriptor of primitive data of type LOB we need to store: the address to one extension in the LOB segment where the contents of the data is stored, the data size and the first 28 bytes of the data content.

For the descriptors of non primitive data we need to store the references to the collection of labeled data corresponding to the content, to achieve this goal we need to store in the CONTENT segment the addresses for: the root of the tree ordered by identifier, the head of the list ordered by identifier, the root of the tree ordered by the labels and the head of the list ordered by labels.

4.2 The CONTENT segment

The CONTENT segment has the responsibility of storing the content of the semi structured data, except for the LOB type data. For the simple primitive type we store strings of extensions that make the content. For the non primitive data we store the collection of labeled semi structured data that make its content.

The implementation of the collection of labeled semi structured data has some point that deserve a commentary. We expect that less than a half of the data in one semi structured data base be of some non primitive type, and most of them with a small quantity of subdata, but there will be a few data with a big amount of subdata, which will be used frequently. Some operations on the database affect directly the content of the data, some others need to search the subdata in the most efficient way, search operations may be invoked by the use of the identifier or the label of the subdata, we may find several subdata with the same label in a given data, also we can find the same data with different labels.

We needed a data structure that can provide this features without loosing its performance levels.

The solution we present in this work is based in the use of a four elements data structure to implement the collections of labeled semi structured data. This structure has two AA-trees and two double linked lists, so we can order the subdata based on the identifier and on the labels; the trees allows the fast search of some data and the lists allow the handling of repetitions and the sequential content recovery. The trees and lists are ordered, one by the identifiers and the other by the labels.

AA-trees [6] are a variation of the black-red trees; AA-trees, black-red trees and AVL trees are balanced binary trees, so its height is $O(\log n)$, particularly AVL trees height is $\log_2 n$, whereas black-red and AA-trees height is $2\log_2 n$. The main problem with AVL and black-red trees is its implementation complexity for the operations, mainly the delete operation. AA-trees have an easier implementation, they are the best option when we need balanced trees with a delete operation, so they are the ideal choice when we have operations on semistructured data. Due to the fact that AA-trees are binary trees we can combine other data structures in its nodes.

In the relational data base systems, B and B^+ trees are the data structures most used sort the registers in a table, this trees are very efficient at organizing vast amounts of data, but they aren't well suited for the semi structured data, whereas AA-trees are, so we can not combine other structures because they aren't binary trees; besides, the nodes of B and B^+ trees usually span the complete page and it is estimated that they have more than half of their space as trash, if we know that almost all non primitive data have few subdata this will make space trashing a big issue.

Within each node in the collection of labeled semistructured data we can put the information needed to keep our four data structures. For each node we must have the left and right pointers of each tree, the pointer to the parent and the height of the node, as well as the previous and next pointer of each list. As we need to manage the four data structures we must have a pointer to the parent of each tree on every node, this is important because after the delete operation we must keep the consistency of the collection, the pointer to the parent is of great help, although its use complicates our algorithm.

Each node in the four data structures is stored in one unit of the CONTENT segment. Each of the four data structures has its own distinguished node, which is used to get into the structure; the nodes are the roots of the trees and the list heads. The addresses of this nodes should be stored in the non primitive data descriptor (in the ID segment) which is part of the collection.

We use the CONTENT segment to store the nodes that are part of the labeled semi structured data collections and the content extensions; this is done for the four data structures. Content extensions are used when the simple primitive data can not be stored into the descriptor, or when the labels don't fit completely in the nodes of the four data structures. The extensions and the nodes are stored in units of the CONTENT segment, whose size is 200 bytes.

4.3 The LOB segment

Intuitively, large objects (LOBs) are those whose size is so big that is very difficult to manage them in main memory. There are two kinds of LOBs: BLOBs (byte oriented LOBs) and CLOBS (character oriented). LOBs are handled as if they were random access files, with mechanisms to read or write in some specific location. Images, videos, music and large texts are examples of large objects.

To take advance of the space, the unit size of the LOB segment is the same that the page's size used by the manager system. Given P the page size and $D = [p/12]$, D is the number of addresses can be stored in a page.

For storing the content of a LOB 5 kinds of units are used (fig. 5): Data unit for to storage a piece of the object's content, P bytes; indirect unit which contents D addresses of data units; double indirect unit which contents D addresses of indirects units; triple indirect unit which contents D addresses of double indirects units; and index unit which contents $D-4$ addresses of data units, a indirect unit, a double indirect unit, a triple indirect unit and an index unit.

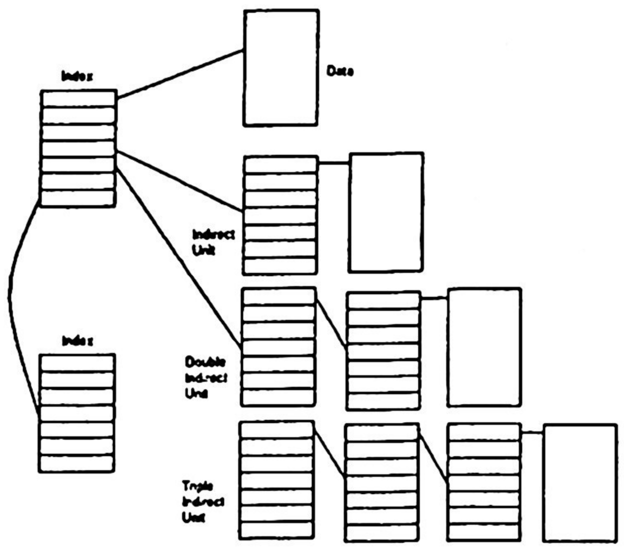

Fig. 5. Organization for storage of LOBs.

Using a single index unit can address a content size of $(D-4)P + DP + D2P + D3P$ bytes, in the table 2 are shown the values for distinct page sizes. If the space isn't enough chains of index units can be created. The LOB descriptor points to the first index unit of the LOB content.

4.4 The PARENTS segment

The PARENTS segment stores the collections of parents owned by the semistructured data, its expected that the most of the data has only one parent or a small

Table 2. Page sizes and LOB space addressable by an index unit.

Page size	Address size
128 B	184 KB
256 B	2 MB
512 B	40 MB
1 KB	607 MB
2 KB	10 GB
4 KB	152 GB
8 KB	2 TB
16 KB	38 TB
32 KB	607 TB
64 KB	9 PB
128 KB	152 PB
256 KB	2 EB
512 KB	38 EB
1 MB	607 EB
2 MB	9 ZB
4 MB	152 ZB

quantity of them. To know the parents' collection is important for improving a better efficiency and keep the data consistency.

For implementing the parents' collections also is used an AA-Tree and a double linked list, this avoids the excessive trashing that other structures provides. For parents' collections is only required a double structure ordered by identifier because only the parents' identifier are needed. The units of the PARENTS segment are a combination of a AA-Tree node ordered by identifier and a double linked list ordered by the same criterion. The root of the AA-Tree and the head of the list are stored in the descriptor of the datum in the ID segment. The PARENTS segment's units have a size of 73 bytes.

4.5 The DICTIONARY segment

The data dictionary storages the names of the ssd-table and the identifiers of their roots, the data dictionary can see as an a collection of labeled semistructured data, the label corresponds to the ssd-table name and it can be repeated.

The DICTIONARY segment stores a descriptor of the labeled semistructured data collection, the nodes for building the collection and the extensions for the ssd-tables' names. In the DICTIONARY segment we also use a four element data structure, this structure has two AA-Trees and two double linked list to order the structure for identifier and name.

The units in the DICTIONARY segment has a size of 200 bytes, like the units in the CONTENT segment, the differences that the CONTENT segment stores several collections and the DICTIONARY segment stores only one COLLECTION.

5 Conclusions

In this work we presented the main aspects of the design of the storage manager for semistructured data bases. The storage manager was built in three levels: Primitive storage, advanced storage and database storage. In the primitive storage is defined the page size (this is the minimum read-write unit that the manager will use), is incorporated the intermediate memory manager, provides the grouping of multiple devices and the using of raw and cooked partitions. In the advanced storage we provides a storage model in three organization levels: Segment, space and unit. In the advanced storage is where the free space is organized.

In the database storage semistructured data are transformed into specialized structured like AA-trees, lists, indexes, etc. to be managed by the advanced storage and implement their functionality.

We reach an native semistructured data storage, because the construction was based in basic computer units without the use of other database models or paradigms. The storage system has a good performance and can be used in applications requiring big quantities of information, it can be concluded from the storage capabilities shown in the tables 1 and 2.

The design of the storage manager was a complicated process because the great quantity of details to consider, the implementation was a delicate process, but can be successfully concluded.

In the future works derived from this work are: the optimization of the source code, the developing of tools for tuning and administration, the construction of concurrency manager, recovery system and backup system, all those aiming to complete a SSDBMS. system.

References

1. A. Silberschatz, H. Korth & S. Sudarshan. Database System Concepts. McGraw-Hill. 4th. edition, 2002.
2. R. Ramakrishnan & J. Gehrke. Database Management Systems. McGraw-Hill. 3rd. edition, 2003.
3. H. Garcia Molina, J.D. Ullman & J. Widom. Database System Implementation. Prentice Hall, 2000.
4. A. Tanenbaum. Modern Operating Systems. Prentice Hall. 1992.
5. A. Silberschatz, P. Galvin & Greg Gagne. Operating Sistems. Limusa Wiley. 6th edition, 2002.
6. M. Weiss. Data structured in Java. Addison Wesley. 2000.
7. Overview of IBM Informix Dynamic Server. International Business Machines Corporation, 2001.