Important notes

B+ Tree

A B+ tree is a rooted tree satisfying the following properties.

All paths from root to leaf are of the same length

Each node that is not a root or a leaf has between [N/2] and n children.

A leaf node has between [(n-1)/2] and n-1 values.

Special cases:

If the root is not a leaf, it has at least 2 children.

If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n-1) values.

Number of pointers / keys for b+ tree

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max ptrs | Max keys | Min ptrs -> data | Min key |
| Non-leaf  (non-root) | N+1 | n | Γ (n-1) / 2 [File:Latin capital letter reversed Ghe.svg](https://upload.wikimedia.org/wikipedia/commons/d/d4/Latin_capital_letter_reversed_Ghe.svg) | Γ (n+1) / 2 [File:Latin capital letter reversed Ghe.svg](https://upload.wikimedia.org/wikipedia/commons/d/d4/Latin_capital_letter_reversed_Ghe.svg) |
| Leaf  (non - root) | N+1 | N | L (n+1) /2 ⅃ | L (n+1) /2 ⅃ |
| Root | N+1 | N | 1 | 1 |

Bulk Storage and file system records

Buffer Management

Index allocation

Index access

hash index

B+ tree

B tree

Ema insertion, delete aa badhu

bitmap disk storage

**Book – 1**

(database-management-systems-raghu-ramakrishnan.pdf)

Buffer Management

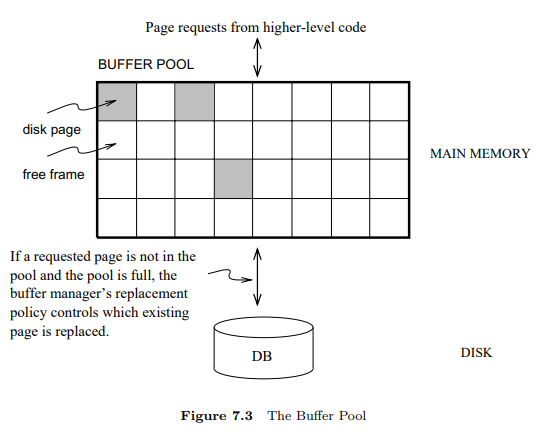
* Some explanation in ppt : - buffer Manager Ptr Swizzling 2022

What is Buffer management ?

To understand the role of the buffer manager, consider a simple example. Suppose that the database contains 1,000,000 pages, but only 1,000 pages of main memory are available for holding data. Consider a query that requires a scan of the entire file. Because all the data cannot be brought into main memory at one time, the DBMS must bring pages into main memory as they are needed and, in the process, decide what existing page in main memory to replace to make space for the new page. The policy used to decide which page to replace is called the replacement policy.

In terms of the DBMS architecture presented in Section 1.8, the buffer manager is the software layer that is responsible for bringing pages from disk to main memory as needed. The buffer manager manages the available main memory by partitioning it into a collection of pages, which we collectively refer to as the buffer pool. The main memory pages in the buffer pool are called frames; it is convenient to think of them as slots that can hold a page (that usually resides on disk or other secondary storage media).

Higher levels of the DBMS code can be written without worrying about whether data pages are in memory or not; they ask the buffer manager for the page, and it is brought into a frame in the buffer pool if it is not already there. Of course, the higher-level code that requests a page must also release the page when it is no longer needed, by informing the buffer manager, so that the frame containing the page can be reused. The higher-level code must also inform the buffer manager if it modifies the requested page; the buffer manager then makes sure that the change is propagated to the copy of the page on disk. Buffer management is illustrated in Figure 7.3.



In addition to the buffer pool itself, the buffer manager maintains some bookkeeping information, and two variables for each frame in the pool: pin count and dirty. The number of times that the page currently in a given frame has been requested but not released—the number of current users of the page—is recorded in the pin count variable for that frame. The boolean variable dirty indicates whether the page has been modified since it was brought into the buffer pool from disk.

Initially, the pin count for every frame is set to 0, and the dirty bits are turned off. When a page is requested the buffer manager does the following:

1. Checks the buffer pool to see if some frame contains the requested page, and if so increments the pin count of that frame. If the page is not in the pool, the buffer manager brings it in as follows: (a) Chooses a frame for replacement, using the replacement policy, and increments its pin count. (b) If the dirty bit for the replacement frame is on, writes the page it contains to disk (that is, the disk copy of the page is overwritten with the contents of the frame). (c) Reads the requested page into the replacement frame.
2. Returns the (main memory) address of the frame containing the requested page to the requestor.

Incrementing pin count is often called pinning the requested page in its frame. When the code that calls the buffer manager and requests the page subsequently calls the buffer manager and releases the page, the pin count of the frame containing the requested page is decremented. This is called unpinning the page. If the requestor has modified the page, it also informs the buffer manager of this at the time that it unpins the page, and the dirty bit for the frame is set. The buffer manager will not read another page into a frame until its pin count becomes 0, that is, until all requestors of the page have unpinned it.

If a requested page is not in the buffer pool, and if a free frame is not available in the buffer pool, a frame with pin count 0 is chosen for replacement. If there are many such frames, a frame is chosen according to the buffer manager’s replacement policy. We discuss various replacement policies in Section 7.4.1.

When a page is eventually chosen for replacement, if the dirty bit is not set, it means that the page has not been modified since being brought into main memory. Thus, there is no need to write the page back to disk; the copy on disk is identical to the copy in the frame, and the frame can simply be overwritten by the newly requested page. Otherwise, the modifications to the page must be propagated to the copy on disk. (The crash recovery protocol may impose further restrictions, as we saw in Section 1.7. For example, in the Write-Ahead Log (WAL) protocol, special log records are used to describe the changes made to a page. The log records pertaining to the page that is to be replaced may well be in the buffer; if so, the protocol requires that they be written to disk before the page is written to disk.)

If there is no page in the buffer pool with pin count 0 and a page that is not in the pool is requested, the buffer manager must wait until some page is released before responding to the page request. In practice, the transaction requesting the page may simply be aborted in this situation! So pages should be released—by the code that calls the buffer manager to request the page—as soon as possible.

A good question to ask at this point is “What if a page is requested by several different transactions?” That is, what if the page is requested by programs executing independently on behalf of different users? There is the potential for such programs to make conflicting changes to the page. The locking protocol (enforced by higher-level DBMS code, in particular the transaction manager) ensures that each transaction obtains a shared or exclusive lock before requesting a page to read or modify. Two different transactions cannot hold an exclusive lock on the same page at the same time; this is how conflicting changes are prevented. The buffer manager simply assumes that the appropriate lock has been obtained before a page is requested.

* Buffer Replacement policy

The policy that is used to choose an unpinned page for replacement can affect the time taken for database operations considerably. Many alternative policies exist, and each is suitable in different situations.

The best-known replacement policy is least recently used (LRU). This can be implemented in the buffer manager using a queue of pointers to frames with pin count 0. A frame is added to the end of the queue when it becomes a candidate for replacement (that is, when the pin count goes to 0). The page chosen for replacement is the one in the frame at the head of the queue.

A variant of LRU, called clock replacement, has similar behavior but less overhead. The idea is to choose a page for replacement using a current variable that takes on values 1 through N, where N is the number of buffer frames, in circular order. We can think of the frames being arranged in a circle, like a clock’s face, and current as a clock hand moving across the face. In order to approximate LRU behavior, each frame also has an associated referenced bit, which is turned on when the page pin count goes to 0.

The current frame is considered for replacement. If the frame is not chosen for replacement, current is incremented and the next frame is considered; this process is repeated until some frame is chosen. If the current frame has pin count greater than 0, then it is not a candidate for replacement and current is incremented. If the current frame has the referenced bit turned on, the clock algorithm turns the referenced bit off and increments current—this way, a recently referenced page is less likely to be replaced. If the current frame has pin count 0 and its referenced bit is off, then the page in it is chosen for replacement. If all frames are pinned in some sweep of the clock hand (that is, the value of current is incremented until it repeats), this means that no page in the buffer pool is a replacement candidate.

The LRU and clock policies are not always the best replacement strategies for a database system, particularly if many user requests require sequential scans of the data. Consider the following illustrative situation. Suppose the buffer pool has 10 frames, and the file to be scanned has 10 or fewer pages. Assuming, for simplicity, that there are no competing requests for pages, only the first scan of the file does any I/O. Page requests in subsequent scans will always find the desired page in the buffer pool. On the other hand, suppose that the file to be scanned has 11 pages (which is one more than the number of available pages in the buffer pool). Using LRU, every scan of the file will result in reading every page of the file! In this situation, called sequential flooding, LRU is the worst possible replacement strategy.

**Buffer management in practice**: IBM DB2 and Sybase ASE allow buffers to be partitioned into named pools. Each database, table, or index can be bound to one of these pools. Each pool can be configured to use either LRU or clock replacement in ASE; DB2 uses a variant of clock replacement, with the initial clock value based on the nature of the page (e.g., index nonleaves get a higher starting clock value, which delays their replacement). Interestingly, a buffer pool client in DB2 can explicitly indicate that it hates a page, making the page the next choice for replacement. As a special case, DB2 applies MRU for the pages fetched in some utility operations (e.g., RUNSTATS), and DB2 V6 also supports FIFO. Informix and Oracle 7 both maintain a single global buffer pool using LRU; Microsoft SQL Server has a single pool using clock replacement. In Oracle 8, tables can be bound to one of two pools; one has high priority, and the system attempts to keep pages in this pool in memory.

Beyond setting a maximum number of pins for a given transaction, there are typically no features for controlling buffer pool usage on a per-transaction basis. Microsoft SQL Server, however, supports a reservation of buffer pages by queries that require large amounts of memory (e.g., queries involving sorting or hashing).

Other replacement policies include first in first out (FIFO) and most recently used (MRU), which also entail overhead similar to LRU, and random, among others. The details of these policies should be evident from their names and the preceding discussion of LRU and clock.

* Power Swizzling

In some applications, objects are retrieved into memory and accessed frequently through their oids; dereferencing must be implemented very efficiently. Some systems maintain a table of oids of objects that are (currently) in memory. When an object O is brought into memory, they check each oid contained in O and replace oids of in-memory objects by in-memory pointers to those objects. This technique is called pointer swizzling and makes references to in-memory objects very fast. The downside is that when an object is paged out, in-memory references to it must somehow be invalidated and replaced with its oid.

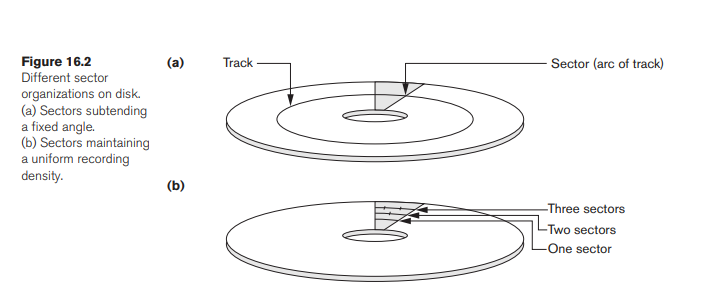
**Book : - 2**

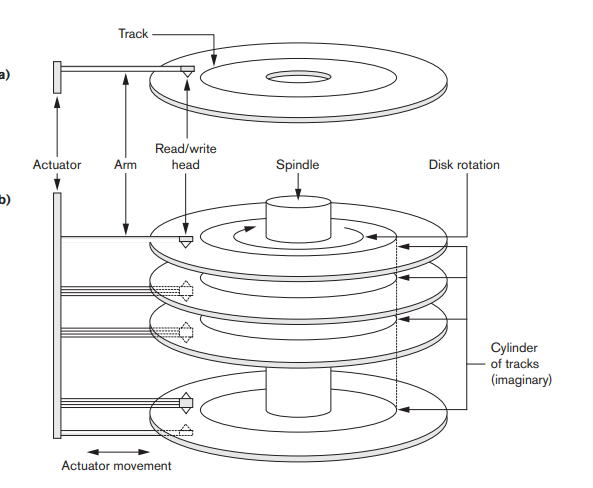
(Ramez Elmasri, Shamkant B. Navathe - Fundamentals of Database Systems-Pearson (2015).pdf)

Disk Storage

Magnetic disks are used for storing large amounts of data. The device that holds the disks is referred to as a hard disk drive, or HDD. The most basic unit of data on the disk is a single bit of information. By magnetizing an area on a disk in certain ways, one can make that area represent a bit value of either 0 (zero) or 1 (one). To code information, bits are grouped into bytes (or characters). Byte sizes are typically 4 to 8 bits, depending on the computer and the device; 8 bits is the most common. We assume that one character is stored in a single byte, and we use the terms byte and character interchangeably. The capacity of a disk is the number of bytes it can store, which is usually very large. Small floppy disks were used with laptops and desktops for many years—they contained a single disk typically holding from 400 KB to 1.5 MB; they are almost completely out of circulation. Hard disks for personal computers currently hold from several hundred gigabytes up to a few terabytes; and large disk packs used with servers and mainframes have capacities of hundreds of gigabytes. Disk capacities continue to grow as technology improves.

Whatever their capacity, all disks are made of magnetic material shaped as a thin circular disk, as shown in Figure 16.1(a), and protected by a plastic or acrylic cover. A disk is single-sided if it stores information on one of its surfaces only and double-sided if both surfaces are used. To increase storage capacity, disks are assembled into a disk pack, as shown in Figure 16.1(b), which may include many disks and therefore many surfaces. The two most common form factors are 3.5 and 2.5 inch diameter. Information is stored on a disk surface in concentric circles of small width,5 each having a distinct diameter. Each circle is called a track. In disk packs, tracks with the same diameter on the various surfaces are called a cylinder because of the shape they would form if connected in space. The concept of a cylinder is important because data stored on one cylinder can be retrieved much faster than if it were distributed among different cylinders.





16.3.1 Buffer Management - page 557

Single-Level Ordered Indexes

The idea behind an ordered index is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book. We can search the book index for a certain term in the textbook to find a list of addresses—page numbers in this case—and use these addresses to locate the specified pages first and then search for the term on each specified page. The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a linear search, which scans the whole file. Of course, most books do have additional information, such as chapter and section titles, which help us find a term without having to search through the whole book. However, the index is the only exact indication of the pages where each term occurs in the book.

For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an indexing field (or indexing attribute).1 The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are ordered so that we can do a binary search on the index. If both the data file and the index file are ordered, and since the index file is typically much smaller than the data file, searching the index using a binary search is a better option. Tree-structured multilevel indexes (see Section 17.2) implement an extension of the binary search idea that reduces the search space by two-way partitioning at each search step to an n-ary partitioning approach that divides the search space in the file n-ways at each stage.

There are several types of ordered indexes. A primary index is specified on the ordering key field of an ordered file of records. Recall from Section 16.7 that an ordering key field is used to physically order the file records on disk, and every record has a unique value for that field. If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field— another type of index, called a clustering index, can be used. The data file is called a clustered file in this latter case. Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, but not both. A third type of index, called a secondary index, can be specified on any nonordering field of a file. A data file can have several secondary indexes in addition to its primary access method. We discuss these types of single-level indexes in the next three subsections.

17.1.1 Primary Indexes – page – 603

All indexes

**Book - 3**

elektroModulatorDva.pdf

B-trees (page – 634) insert delete methods

A B-tree organizes its blocks into a tree that is balanced, meaning that all paths from the root to a leaf have the same length. Typically, there are three layers in a B-tree: the root, an intermediate layer, and leaves, but any number of layers is possible. To help visualize B-trees, you may wish to look ahead at Figs. 14.11 and 14.12, which show nodes of a B-tree, and Fig. 14.13, which shows an entire B-tree.

There is a parameter n associated with each B-tree index, and this parameter determines the layout of all blocks of the B-tree. Each block will have space for n search-key values and n + 1 pointers. In a sense, a B-tree block is similar to the index blocks introduced in Section 14.1.2, except that the B-tree block has an extra pointer, along with n key-pointer pairs. We pick n to be as large as will allow n -1-1 pointers and n keys to fit in one block.

Bitmap Indexes (page- 688)

15.7 Buffer Management (page - 746)

**Book - 4**

( Database-System-Concepts-etc.pdf )

File Organization (page : 588)

Heap File Organization (page : 596)

Buffer Manager ( page - 604)

Indexing (All Index ) ( page – 623 and page - 634)

B tree and B+ tree (page : - 655)

Hash Indices (page : - 658)