Overview of Data Storage in RDBMS

Physical Characteristics of Disks

A *disk drive* consists of

- disk assembly: part of the disk drive that contains the physical data storage.
- head assembly: part of the disk drive that contains *heads* for accessing data.

Disk assembly consists of disk platters. Each platter has two surfaces.

Disk surfaces are partitioned into circular *track*, each consisting of a number of *sectors*.

A collection of tracks equidistant from the center on all surfaces is called a *cylinder*. (in other words: the tracks that are under the disk head at the same time).

Disk Access

A disk **block** or **page** is a logical unit of data storage which ***can be read from disk in a single disk access command***.

Often a disk block is equal in size to a disk sector, or a small number of disk sectors (2,3,4).

Disk latency: time between a disk access command is issued and the data is delivered from disk to main memory. Disk latency consists of the following components:

- disk controller processing time (typically, fraction of a ms.);
- *seek time*: time it takes to move disk head to the correct cylinder (10-40ms);
- *rotational latency*: time it takes to rotate the disk (10ms for a full rotation);
- *transfer time*: the time it takes to read in all the sectors of the block. ($\sim$ 10Mb/sec).

I/O model of Computation

Observations:

- In database applications, the amount of data exceeds main memory capacity of the computers.
- When data is stored in secondary storage (on disks), **data access time dominates all other computation times during query execution**.

Conclusions:

- Efficiency of query processing in DBMS needs to be measured w.r.t. disk access it requires, rather than w.r.t. computational complexity of the algorithms.
- It is of utmost importance to implement efficient disk access in DBMS.
- Efficient disk access can be implemented as follows:
  - *Confine all disk access in DBMS to* ***block/page*** *access.* That is, all disk access commands in DBMS must access a full disk block.
  - Develop and implement techniques for efficient storage and access data stored in disk blocks.

To ensure that data stored in databases is retrieved efficiently, DBMS choose the following data storage approach:

- A **block** or **disk page** of a specific size is selected. The size of a block cannot be too large — this may lead to wasted space, but should not be too small — this will increase the number of disk access operations.
- **Each relational table, and all supplemental index structures are stored as collections of blocks/pages on disk**.

Standard block sizes are 2, 4, 8, 16Kb. Larger blocks are used less often.

Data Storage Techniques

Cylinder-based Organization. Store consecutive blocks on a single cylinder (on different surfaces). Improves seek time for consecutive read operations. But if access requests are "random", will not help.

Use of Multiple Disks. Multiple disks mean multiple disk controllers, which means, more disk access requests per time unit can be satisified. But requires extra investment in the disk infrastructure.

Mirroring. Here, multiple disks are used again, each repeating the data stored on other disks. This improves the response times and the number of requests processed per time unit, but has higher cost and poses issues when data is updated.

Elevator Algorithm. If multiple disk access requests come to disk controller at the same time, we can schedule them using the same principles as are used in elevator operation - assuming the disk head is our "elevator" and tracks are "floors". This can streamline processing of multiple requests, but is not as efficient for non-busy request schedules.

Prefetching/Double Buffering. Sometimes it is possible to "know" which blocks will be needed and schedule their optimal retrieval before the actual access requests arrive. But, prefetching requires extra main memory.

Storing Relational Data in Disk blocks.

Note, the the solution proposed below is by far, not the only possible. There may be a lot of different ways to store relational data on disk, e.g., grouping attributes together, using **only** index structures, storing data from different tables in the same file, etc... As we discuss the traditional storage techniques, it will become apparent why they are used and are considered efficient.

Each relational table is stored on disk as a single file, broken into a sequence of disk pages. Individual blocks may be located in different places on disk (different surfaces, tracks, cylinders, etc. . .), but all content within a single page is a consequtive seqence of bytes.

A *disk block* is typically equal to one or more *sectors* of the disk. Note that *sector* is a term that describes physical properties of the disk, while *block* and *page* describe logical constructs.

A typical **relational table file** consists of

1. Header page, the first page of the file.
2. Data pages, all remaining pages in the file.

Occasionally, depending on the types of data stored in the relational table, other types of disk pages may be present in the file as well. The structure of a relational table file is shown on Figure 1.

Header page

A header page is the first page/block of any database file. This page does not contain any relational data from the database. Rather it contains useful information about the file itself, as well as meta-information about the relational table.

The following information can be stored on the header page:

- Format identification information (something that tells DBMS, "I am your file");
- Relational table schema information;
- Record structure information: sometimes record structure on disk is different than the logical structure of the table.
- Record size;

An overview of a Relational Table File structure

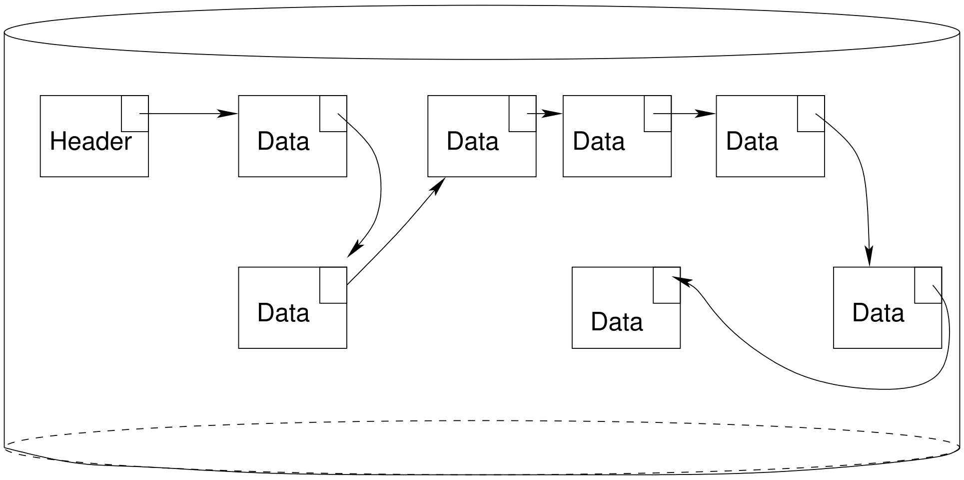

Figure 1: Database files on disk.

- Starting points/pointers for various linked lists within the table file:
  - Full ordered list of blocks;
  - List of blocks with open space;
- Indexing information for *non-heap* database files (we will look at how records are organized in the file below);
- Various timestamps.
- etc...

Generally speaking, the size of the block is typically more then enough to accommodate any information DBMS designer finds useful to store in header pages.

Data page

Figure 2 shows the structure of a data block. It consists of

1. *Block header*, a sequence of bytes (typically at the beginning of the page) that stores information about the current state of the disk page and any linked list pointers for the entire file.
2. *Data records*: the main portion of the disk page is broken into *records*, sequences of bytes storing data from a single row or the relational table.
3. *unused space*: any leftover space that cannot be used to store a full record.

Storing Data in Disk Records

Each record stores information about a single tuple. It is broken into parts representing values of each individual attribute of the tuple. In addition, records may contain some extra information.

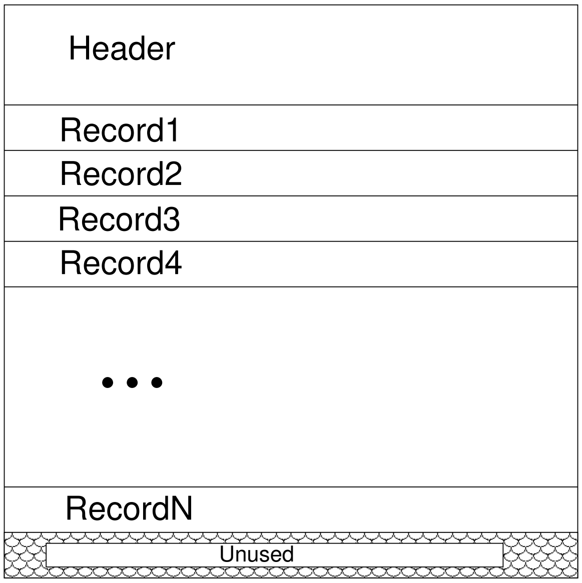

Figure 2: Data Page/Block structure in a nutshell

Representing Attributes

Attribute Type	Storage Requirements
`INTEGER`	2 or 4 bytes
`FLOAT`	4 or 8 bytes
`CHAR(n)`	array of n bytes; unused bytes marked with $\perp$ (“pad”) character
`VARCHAR(n)`	*Length+content*: array of $n + 1$ bytes; first byte holds # bytes in string, remaining bytes hold string content; *Null-terminated string*: array of $n + 1$ bytes, filled with string characters, terminated by *null* character
`BIT(n)`	array of $(n\texttt{div}8) + 1$ bytes
Enumerated types	Map values to integers, store as `INTEGER` value
`DATE, TIME`	Converted into `INTEGER`

Grouping Attributes in Records

Most DBMS use records of *fixed size*. Here we will concentrate on such records. Other applications, e.g., Information Retrieval, require records of varying size. These will be discussed separately later.

Building Fixed-size Records

First approach to building a record is to *concatenate the represenations of the tuple's attributes (a.k.a., fields) together*. However, in doing so, the following rules **must be observed**:

- *INTEGER* values (for 4-byte integers) must start at positions divisible by 4.
- *FLOAT* values (for 8-byte floating point numbers) must start at positions divisible by 8.

- Sometimes, the rule is that all other fields must start at positions divisible by 4.

If there are fields of sizes not divisible by 4, or if concatenation leads to INTEGERs and FLOATs starting at wrong positions, the following can be done to rectify the situation:

- Padding: empty, unused bytes are added between the fields to ensure that the next field starts at the right position/offset.
- Field reodering: the record is built by fist putting all FLOAT values, then all INTEGER values, then all other values. Because this structure may differ from the order of attributes specified by `CREATE TABLE` statement, the new order of attributes needs to be recorded somewhere (e.g., in the header page of the file).

Record Headers

In addition, records can contain *headers*. The choice of whether to include a record header, and what information to put in it is up to the DBMS designer. Record headers for fixed-size records can contian the following information:

- Record schema/pointer to the place in the file where record schema is stored (may also be contained in the block header).
- Length of the record.
- Timestamps for record modification/access
- Record state (active/deleted), a.k.a, "tombstone".

Block Headers

Block Headers typically store information about the current state of the block, and the block's "position" in the overall file. The information stored in the header is determined by the DBMS designer. Typically, it may include the following:

- Block ID;
- Links to other blocks in the file:
  - next/previous block in the block order in the file;
  - next/previous block that has space avaialble for new records;
  - next/previous block in an special indexing order;
- Information about the relation, relational schema/pointer to the schema.
- General information about records in the block: record size, total number of slots, number of used records, etc...
- Information about availble record space on the block:

- Number of available slots/records;
- Record availability bitmap;

- Timestamps.

Maintenance of Data Stored on Disk

Data Organization in a File

Individual records can be organized in a number of different ways in the database file.

Heap File Organization. Heap File denotes a record organization method where records are stored on disk pages on first-come — first-serve basis, in no particular order. That is, any record can be placed anywhere in the file, subject to space availability.

Sequential File Organization. A *search key* is defined for the relation, and records are stored ordered according to the search key. Note that the *search key* need not be a primary key, or even a superkey of the relation being stored. New records must be inserted according to their search key value.

Hashing File Organization. Records are hashed on some attribute value. Each hash value is associated with a block (sequence of blocks is overflow buckets are needed) where the record is to be stored.

Record Modifications

We need to discuss three basic types of modification:

- Record Insertion
- Record Deletion
- Record Update

Insertion

Record insertion procedures are different for different file organizations.

Insertion into a Heap File

To insert records into a heap file, it is convenient to keep a (double-)linked list of all heap file pages with available space. The pointer to the first page of the list can be stored on the *header page*. The pointers to next/previous page on the list can be stored in the *block headers*.

Assuming existance of such support, insertion of a record into a heap file can be done as follows:

```
Algorithm InsertRecordHeap(File F, Record R)

begin
   HeaderPage = ReadBlock(F,1);   // Retrieve the header page of the file
   FreeSpacePageId = HeaderPage.FreeSpaceList;

   Block = ReadBlock(F,FreeSpacePageId);  // Retrieve the block
   RecordNum = FindFreeSlot(Block);       // find a free slot
   Put(R, Block, RecordNum);              // write the contents of R into the
                                          // available slot
   if (No more free space left in Block)
         begin
           HeaderPage.FreeSpaceList = Block.FreeSpaceListNext;
           NextBlock = ReadBlock(F,Block.FreeSpaceListNext);
   NextBlock.FreeSpaceListPrevious = 1;
   WriteBlock(NextBlock, F, Block.FreeSpaceListNext);
 end

    WriteBlock(Block, F, FreeSpacePageId); // write the data back to disk
    WriteBlock(HeaderPage, F, 1);
 end
```

Notice that this algorithm assumes that `FreeSpacePageId` is not NULL. If it is NULL, then a new page needs to be created, and the records needs to be put on it.

Finding a free slot on the disk page

There is a number of ways by which a free slot can be found on the disk page. All the computations occur in main memory and do not have I/O costs associated with them.

1. Direct Scan. Each record contains a *tombstone* - a byte indicating whether it is active or deleted. The disk page is scanned until the first record with the *tombstone* set to *"deleted"* is found.
2. Bitmap. Record header contains a bitmap, specifying which of the records on the page are available. Instead of the entire disk page, the bitmap is scanned until the first available slot is discovered. After the record is inserted, the bitmap must be updated.

Insertion into a Sequential File.

When data is inserted into a sequential file, the following must be observed:

- The record needs to be inserted according to the value of its key.

- The file needs to be scanned (or an outside index needs to be used) to find the disk block where there record must be inserted.
- If there is no empty slot at the location where the record needs to be inserted, *record sliding* must be used to shift all subsequent records on the page (assuming there is space available). This operation is as simple *byte copy*, so it can be performed fast.
- If there is not space on the page, two possibilities can be considered:
  - **Record sliding to next page.** If the next disk block has enough space, we can do record sliding and put some overflow records on the next page, using record sliding there as well.
  - **Overflow page.** Alternatively, a new disk block can be allocated. The disk block is inserted between the current block and the next block in the *search key* order. The contents of the current block are split roughly evenly between the old block and the new block, after which the new record is inserted in its designated location (which can be either on the old page, or on the new one).

Insertion into a Hashed File.

Insertion into a *hashed file* proceeds in two steps.

Step 1: The hash key of the record being inserted is computed, and the disk page(s) for the hash bucket is/are identified.

Step 2: An empty slot is found on one of the pages of the hash bucket (if not - a new page is added to the hash bucket), and the record is inserted into that slot. The procedure is similar to the one used for a *heap file*.

Deletion

Unlike insertion, methodology of record deletion does not depend on the type of file used to store the relational table.

Some traditional approaches to deletion are:

Deletion with slide: The deleted record creates a "gap" on a disk page. To cover this gap, the records following the gap on the disk page are shifted left, covering the gap, and moving the empty slot to the end of the page.

Advantages: Free space is always located only at the end of the disk page. One pointer in the block header (or simply a count of occupied slots for fixed-length records) is sufficient to keep track of free space within the block.

Disadvantages: The deleted record gets immediately overwritten. This removes the possibility of an easy "undo" action.Additionally, this complicates handling dangling pointers to the deleted record (the pointers will now point at a completely different "live" record).

Deletion without slide: The slot of the deleted record is declared available, and the record itself is marked as deleted (by activating the tombstone flag, for example), but the actual space occupied by the record stays as-is. The record can be later overwrritten, when a new record is inserted in the same space.

Advantages: Simplicity. This method requires little extra actions, save for changing the status of the slot to “available”.

Disadvantages: Gaps in the disk page. This means that more space or more complicated organization is needed to keep track of free space on the page. Possible solutions include a bitmap in the disk header, or a linked list of all available slots: the pointer to the first available slot is stored in the disk header, while all subsequent pointers are stored in the records themsleves.

Deletion without giving up space: The record is marked as deleted, but the space is not freed. No new record can be stored in this space until a major DB reconstruction/reconciliation occurs.

Advantages: Ensures proper handling of all “dangling pointers”.

Disadvantages: The relation grows in size with every insertion. Deletions do not decrease the space alotted for the insertion.

Additionally, deletions may lead to removal of disk pages from the file. There is a number of approaches here.

- “Lazy evaluation.” The disk page is removed only when it contains no live records on it. Disk pages are never merged.
- “Eager evaluation.” Each time a deletion occurs, the system checks whether current disk page can be merged with one of its neighbors. If the answer is yes, such merger is performed. The minimal criterion for merger is the ability to fit all records into the free space available in the two neighboring pages. Other, stricter criteria (e.g., requiring at least a few empty slots available in the resulting blocks) are also possible.

Update

For fixed-length record, *record updates* do not change the storage. The basic algorithm is the same: the block with the desired record is located on disk (via scan, index structure or other means), the disk page is retrieved, the record is found, update it performed, and the block is flushed back to disk.