DBMS File Organization, Indexes

1. Basics of Hard Disks

All data in a DB is stored on hard disks (HD). In fact, all files and the way they are organised (e.g. the familiar tree of folders and sub-folders you can see in any Windows OS) -- everything is stored in the HD. We first understand how data in a file is stored on the HD, or is sent back to the CPU for processing. Most HD's contain one or more disks, or **platters**, coated with magnetic material. A tiny electro-magnetic coil on the **read/write head** is carried by an **arm**, controlled by a special high speed stepper motor. The disk assembly rotates at a constant speed (e.g. 7200 RPM). All data in a file is converted to series of bits (0 or 1), and each bit is stored at a point on the disk (e.g. 0 → magnetised, 1 → demagnetised). Usually the data is stored along points on concentric circles on the surface, called tracks.

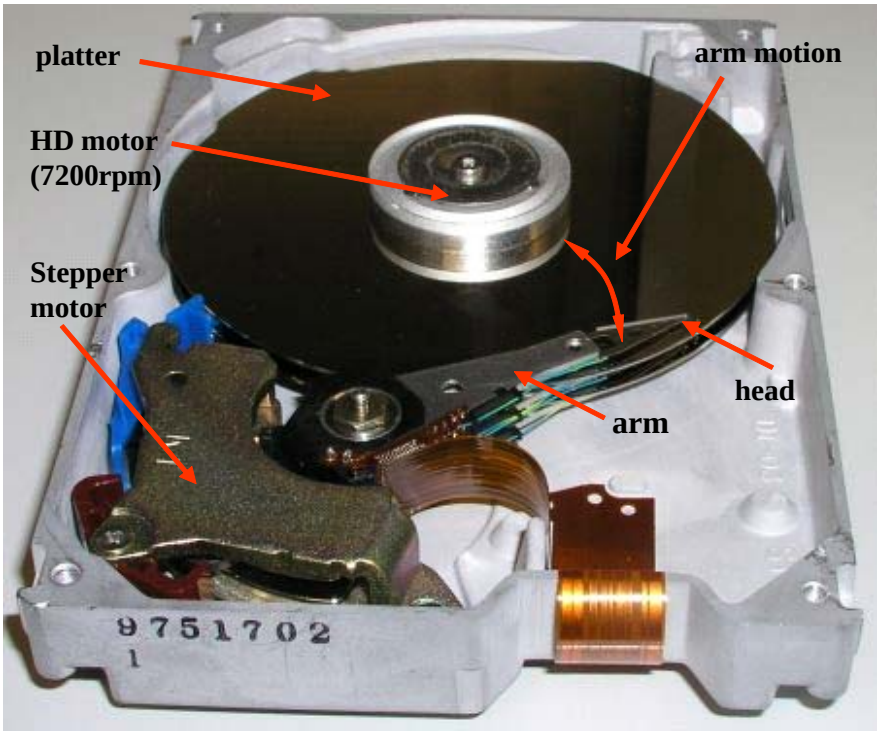

Figure 1. Inside a typical HD; this one has two platters, so there are four magnetic surfaces. The arm assembly has 4 arms, each with a read/write head; the entire arm assembly moves together when activated by the stepper motor, whose motion is controlled by the HD controller.

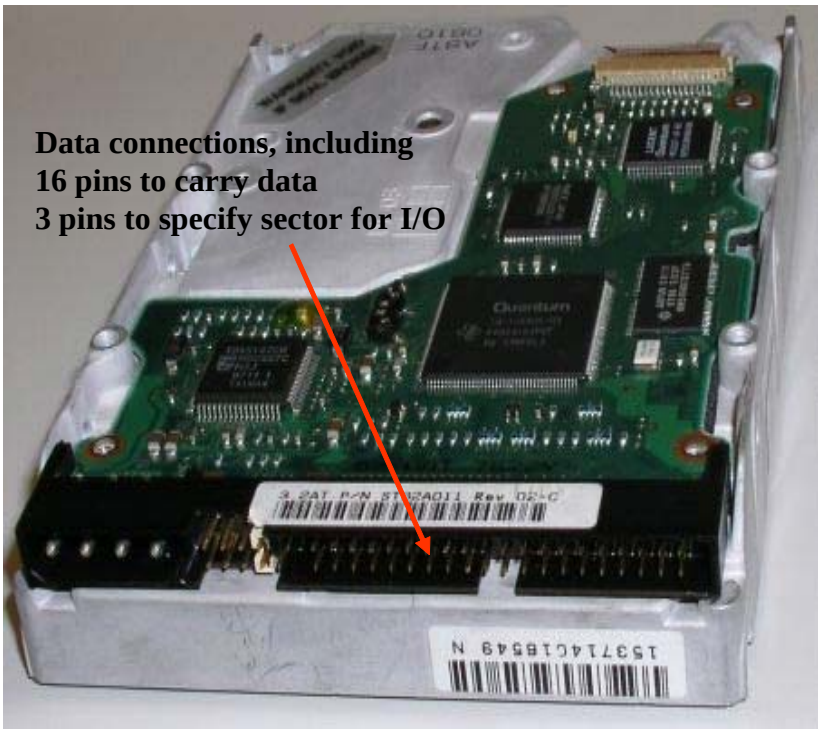

Figure 2. Reverse side of the HD showing a typical IDE HD controller; the 40-pin socket connects to the computer motherboard, and communicates to the CPU.

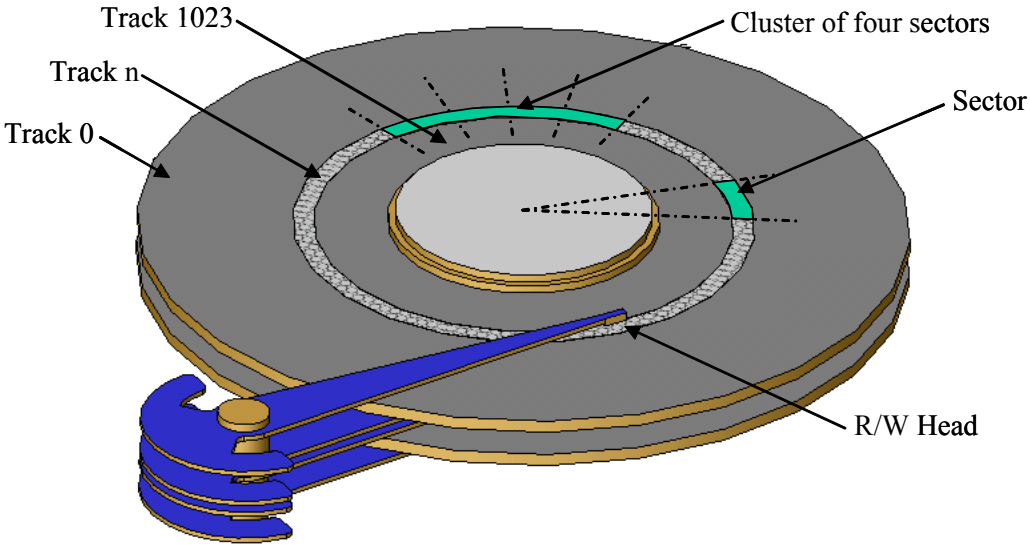

Figure 3. Schematic of data storage on a 1024-track disk: outermost track is labelled Track-0. Smallest unit of data exchange is one sector; group of four sectors is a cluster. All R/W heads are fixed relative to each other on the arm, so four R/W heads can read at the same time -- tracks on different faces/platters that can be read simultaneously make a cylinder.

It is obviously inefficient to transfer data from the HD to the processor one bit at a time; by convention, most HD's communicated in units of 512Bytes, or 0.5 KB. Thus, even if a file is only 10Bytes in sizes, it will be stored in a region on the disk that takes up 512Bytes. This unit is called a **Sector**. Thus all the tracks of the entire HD are broken into chunks, or Sectors. Each side of a typical disk may have 1024 tracks (track 0 is along the outer edge, and track 1023 towards the centre). Each track has an equal number of sectors -- so the distance between two contiguous bits on the outer tracks is larger than on the inner tracks. The HD rotates at constant speed; once the R/W head starts to read a sector, the reading time is the same, irrespective of where the sector is located on the HD. The time taken by the stepper motor to position the R/W head above the correct sector is called **seek time**. Obviously, seek time depends on which was the last sector read by the HD, and also on the speed of the stepper motor. For convenience, we shall refer to the *amount of data in one sector* as **one Block**.

Note that if a file is, say, 800 Bytes in size, then we require two sectors to store it (since $\lceil 800/512 \rceil = 2$). To read the entire file, the HD will work fastest if the two sectors used to store it are neighbouring sectors on the same track. Suppose that this file later increases in size (maybe more data is added to it) to 1900Bytes (needing $\lceil 1900/512 \rceil = 4$ sectors). The optimum way to store it will be to use an additional two *contiguous* sectors on the same track, namely a **cluster** of four sectors (see figure 3). However, if two contiguous sectors are not free, e.g. if they have been used to store a different file, then the HD controller will store the remaining part of the file at a different location, and we say that the file is **fragmented**. In general, more fragmentation means that R/W operations become slower (WHY?).

Usually the tasks of managing information of which sectors are storing what file, and how to organise the data on the disk is handled by the computer OS (e.g. Windows XP) and the HD driver. However, DBMS need to handle special types of files -- since each table can potentially be very large. Therefore most DBMS's will permanently occupy a large portion of the HD, perhaps an entire partition on the HD, and then manage which sector(s) stores what data by directly controlling the HD Driver.

2. Typical DB operations

In the following discussion, we assume that a DB has already been defined, and each table in it already contains some amount (usually large amount) of data. Our concern is: how fast can the DB server respond to some query that is sent to it. When is this important? If a large number of users are simultaneously accessing a DB [Give three real life examples]; and/or if some table that needs to be accessed is very large, namely, has many rows, perhaps millions [Give three real life examples].

The second thing we need to know is "what is the user trying to do with the data in the DB?" In general,

we can divide our analysis into the following types of cases:

- Search for a particular row of data in a table
- Creation a row in a table
- Modify some data in a row of a table
- Deleting a row of data from the table

In terms of the logical operations to be performed on the data, relational tables provide a beautiful mechanism for all of the three above tasks. The storage of a Database in a computer memory is mainly concerned with:

1. The need to store a set of tables [each table can be stored as an independent file];
2. The attributes in a table are often accessed together [Why ?]. Therefore, it makes sense to store the different attribute values in each record contiguously. For each record of a table, the attributes MUST be stored in the same sequence [Why ?]
3. Since there is no prescribed order in which records must be stored in a table, we may choose the sequence in which we store the different records of a table. We shall see that this observation is quite useful.

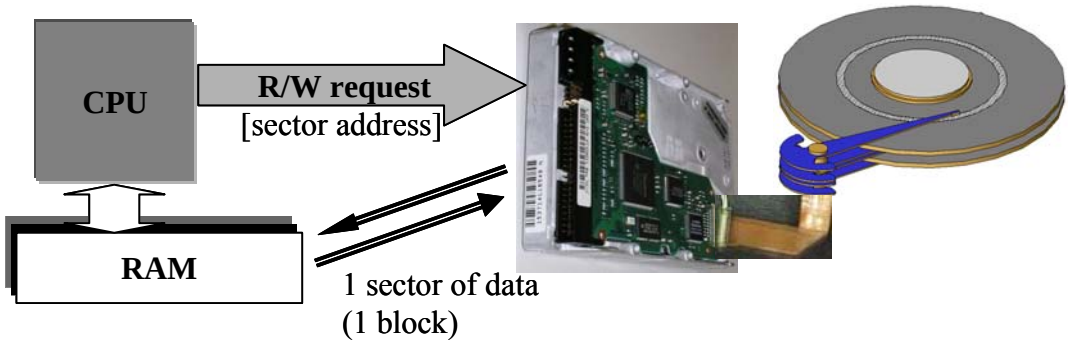

Figure 4. Interaction between CPU and HD takes place by exchanging 1 block at a time.

It is useful to look at **how tables are stored on the HD** in a little more detail. Each row (called **record**) of a table can contain different amount of data [Why?]. Therefore, the record is stored with each subsequent attribute separated by the next by a special ASCII character called a field separator. Of course, in each

block, we may place many records. Each record is separated from the next, again by another special ASCII character called the record separator. For our discussion, we shall assume that a record is stored in a block only if there is sufficient space to store it completely (namely, ***we do not*** store part of the record in one block, and the remaining in the next block). There are some cases where each record of a table can be larger in size than one sector on the HD, for example, a DB storing street maps, where each map segment is a JPEG image of over 3MBytes. However, we will ignore such special cases for our discussion.

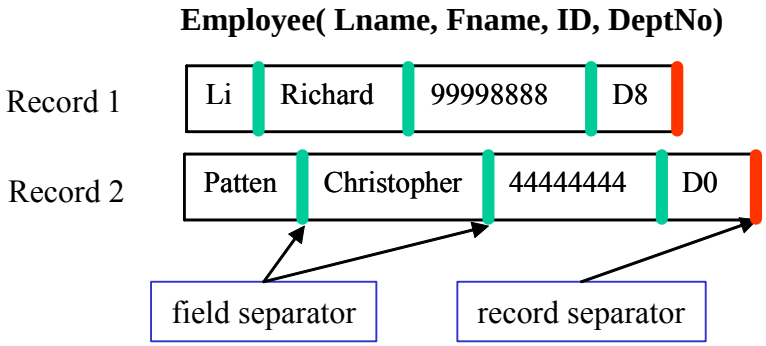

Figure 5. Schematic view of how each record is stored on the HD

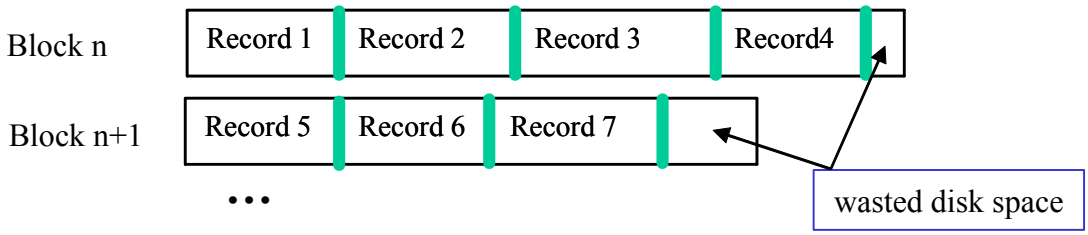

Figure 6. Schematic view of how records are placed per sector on the HD

3. What affects the speed of operations in a DB?

The speed with which a DBMS can execute a query depends on the basic operation that is to be done. Recall that the basic operations are: (i) find a row (ii) insert a row, (iii) delete a row (iv) modify a cell in a row. In order for a computer program to work with the data, it must first read the data to chip memory accessible to the CPU, namely the RAM. In general, the speed at which the CPU and RAM communicate can be 100x or 1000x faster than the speed at which data can be transferred from the HD to the RAM (for example, to fetch one block of data from the HD and load it into the RAM may take 10^{-3}sec, but to search for a record with a particular ID value in one block of data stored in the RAM may take 10^{-6}sec!) [Why?]. Therefore, in all the following analysis, we shall be concerned with the time it takes to transfer ***the required data*** between the HD and RAM to complete an operation.

From figure 4, the time to READ one block of data from the HD requires [time to send the sector address to the HD]+[time to locate the R/W head at the start of the sector]+[time for the head to read the data and put into the HD driver's buffer]+[time to transfer the block of data to the RAM].

Among these, the 2nd component (seek time) and 3rd component (R/W time) dominate. Note that seek time for each block will depend on where the R/W head was located at the end of the previous sector accessed by it; to simplify calculations, hereafter we shall use the ***mean seek time*** for our analysis.

4.1. HEAP files

The simplest method of storing a DB table is to store all the records of the table in the order in which they are created, on contiguous blocks, in a large file. Such files are called **HEAP files**, or a **PILE**. We shall examine the storage methods in terms of the operations we need to perform on the Database.

Operation: Insert a new record

Method: The heap file data records the address of the first block, and the file size (in blocks). It is therefore easy to calculate the last block of the file, which is directly copied into the buffer. The new record is inserted at the end of the last existing record, and the block is written back to the disk.

Performance: Very fast -- in the worst case, the DBMS needs to read two blocks of data and write back one block (i.e. read the last block, if it does not have sufficient empty space, read the next available block and finally, write the record to this block). If the time for transfer of 1 block = t, then the ***worst case time*** required is $2t$. In addition, there is a fixed overhead cost associated with updating the size of the file, which sectors it occupies, etc. -- but this is negligible.

Operation: Search for a record

Method: Linear search: (i) The first block to the RAM; (ii) CPU searches each record in the block to match the search criterion; (iii) If no match is found, copy the next block into RAM and go to (ii).

Performance: Poor. Assume that a table occupies B blocks in size. In the worst case, we will find the data in the last Block (or not find it at all); ***worst case time*** = Bt. On the average, if the searched data is uniformly distributed, the ***average search time*** = $Bt/2$, which is still very poor.

Operation: Update a cell in a record

Method: Linear search: (i) The first block to the RAM; (ii) CPU searches each record in the block to match the search criterion; (iii) If no match is found, copy the next block into RAM and go to (ii); (iv) else, if a record in the block matches the criterion, update the record, and write it back.

Performance: It is easy to see that the ***worst case time*** **=** $Bt+t$, and the ***average time*** = $Bt/2+t$ (assuming that the time to write a block of data = time to read a block = t).

[Note: it is possible that the update increases the size of the block greater than 1 sector; convince yourself that this will introduce a constant, and relatively negligible, overhead]

Operation: Delete a record

Method: Search the record that is to be deleted (requires linear search).

Performance: Poor (analysis is same as for "update a record" case).

Problem: After the record is deleted, the block has some extra (unused) space. What to do about the unused space?

Different DBMS may use a different policy about this. Typical policies may be:

(a) Delete the space and rewrite the block. At periodic intervals (few days), read the entire file into a large RAM buffer and write it back into a new file.

(b) For each deleted record, instead of actually deleting the record, just use an extra bit per record, which is the 'RECORD_DELETED' marker.

IF: RECORD_DELETED == 1, the record is ignored in all searches.

Approach (b) may have some advantage in terms of recovery of data deleted by mistake, or in cases where a deleted record must be recovered. In any case, after fixed intervals, the file is updated just as in case (a), to recover wasted space.

Heaps are quite inefficient when we need to search for data in large database tables.

4.2. Sorted Files:

The simplest method to make efficient searching is to organize the table in a **Sorted File**. The entire file is sorted by increasing value of one of the attributes, called the **ordering attribute** (or **ordering field**). If the ordering field) is also a key attribute, it is called the **ordering key**. Figure 7 shows an example of a table sorted by the "SSN" attribute.

	Lname	SSN	Job	Salary
Block 1	Abbot	1001		
	Akers	1002		
	Anders	1008		
Block 2	Alex	1024		
	Wong	1055		
	Atkins	1086		
Block 3	Arnold	1197		
	Nathan	1208		
	Jacobs	1239		
Block 4	Anderson	1310		
	Adams	1321		
	Aaron	1412		
Block 5	Allen	1413		
	Zimmer	1514		
	Ali	1615		

Block n	Acosta	2085		

Figure 7. A table (only part of the data is shown) sorted by ‘SSN’ as ordering attribute

Operation: Search for a record with a given value of the ordering attribute.

Method: Binary search

For a file of *b* blocks, look in the block number $\lceil b/2 \rceil$

If the searched value is in this block, we are done;

If the searched value is larger than the last ordering attribute value in this block,

repeat the binary search in blocks between ($\lceil b/2 \rceil + 1$) and *b*;

otherwise repeat the search in blocks between 1 and ($\lceil b/2 \rceil - 1$).

Performance: Let us consider the worst case time spent for a sorted file of size *b* blocks. In each iteration, we need to check one block (= t units of time); Worst case ➔ we do not find the record in until the last remaining block. In the 2nd iteration, at most b/2 blocks will remain to be searched; in the 3rd iteration, $(b/2)/2 = b/2^2$ blocks remain. After i-iterations, $b/2^{i-1}$ blocks remain to be searched. Of course, if only one block remains to be searched, no further sub-division of the list of blocks is required, namely, we stop when $b/2^{i-1} = 1$, which gives: $b = 2^{i-1}$, or $i = (1 + \lg_2 b)$. Total number of blocks searched after i-iterations is i, and the total time is $t(1 + \lg_2 b)$.

Let's compare the relative worst-case search time between heap storage and sorted file storage. Let's assume a file occupies 8192 blocks. Stored as a heap file, the worst case time is 8192t; stored as a sorted file, the worst case time is $t(1 + \lg_2 8192) = t(1+ 13) = 14t$. The search is 8192/14 ≈585 times faster!

Operation: Delete a record/update a value in a record (other than the ordering attribute value)

Method: First, search the record to be deleted (using binary search).

After the record is deleted, we still need to perform the usual file compacting once every few days.

Performance: Quite fast. The worst case time is the same as that for search, plus 't' units of time to write the modified block.

Operation: Insert a new record/update a record where the ordering attribute is changed

Method: If we insert the new record in its correct ordered position, we would need to shift every subsequent record in the table! This is of course very time consuming. Consider the case that a new record

is inserted for 'SSN=1003' in our example of figure 7. Since this record goes into the first block, the record for 'SSN=1008' will need to be removed to make space for the new record. The 'SSN=1008' record must be shifted to Block-2, but this may require 'SSN=1086' to be shifted out of Block 2, and so on. Total worst-case time spent is approximately ≈

Search for the insertion point **≈** $t(1+ \lg_2 b)$ + (Read and Write each block = $2bt$)

Performance: Very inefficient.

Solution: **Overflow file**. The table is stored as a sorted file. However, when a new record is inserted, it is stored in a different file, called an overflow file (thus each table has its own main file and its own overflow file). The main file is sorted, while the overflow file is a heap file. *Periodically*, the overflow file is merged with the ordered file.

Thus every insert operation is very fast (constant time) -- just add the record at the end of the overflow file. For other operations, we first use binary searching in the regular file. If we cannot find a matching record, we then perform a linear search in the overflow file. Note that since the **overflow file is not ordered**, it requires a linear search.

4.3. FASTER searching: Hashing Functions

Hashing is a common idea that divides the total data to be stored into a series of organized "buckets". Let's look at a simple, physical example first. Suppose you need to store paper files for several hundred people in file folders. An efficient way to do so is to use 26 folders -- one for all last-names that begin with 'A', one for 'B' etc. Here, each folder can be though of as a 'bucket' storing a subset of the data. The same idea can be used in storing records of a table in a DB file.

In order for Hashing to work, we must first have a reasonable idea of the total size of the data (e.g. the expected maximum number of records, and size of each record). We also decide upon the approximate size of each 'bucket' -- thus, if we expect the maximum size of our table to be 10,000 blocks of data, and we would want a response for a search in under *10t* units of time, then we may use 1000 buckets of 10 blocks

each. Secondly, we must specify the **hashing attribute,** whose value will be used to search for the data -- this attribute will be used to create the '**hashing index**'. Finally, we need to define a **hashing function**: its input is the value of the hashing attribute, and its output is an integer that is in the range [1, …, max number of buckets]. It is beneficial to design hashing functions whose output is uniformly distributed over this range. A simple hash function is:

Hash Address = K mod M,

where K is the **hashing key**, an integer equivalent to the value of the hashing attribute,

M is the maximum number of buckets needed to accommodate the table.

Suppose each bucket has a size of S blocks. Initially, the DBMS will reserve a space of MxS contiguous blocks on the HD. Even if the table has very few records, this entire space is reserved for the hashed table. For simplicity, let's assume that the table is stored in blocks [1, …, MxS]. The first bucket occupies blocks (1, … S), the 2nd bucket is stored from blocks (S+1, …, 2S), etc.

Operation: Insert a record.

Procedure:

1. Using the Hashing attribute value, calculate hashing key, K;
2. Calculate hashing address, [in our example, (K mod M)];
3. Place the record in first sector with enough space, starting from address Sx(K mod M)

Performance: In general, the maximum time that it will take to do an insertion will be $tS + t$ (tS to read the S blocks in this bucket, and t units to write the block where the record is inserted).

NOTE: It is possible that the hashing key values are distributed in such a way that too many records get the same hashing address, such that all S blocks in this bucket are filled but still some records remain unwritten. An analogy could be our physical file example, where we keep 26 folders, one for each starting alphabet of the last-name. However, suppose we have papers of 3000 people, but all of them have last name 'Wong' -- thus our 'W' folder will have no space, while other folders remain empty.

In step 3 of our procedure above, in such cases, we merely store the overflow records in the next hashing address with available space. In general, such an 'overflow' is not frequent; therefore in general, the

performance of hashed search is very good indeed (irrespective of the size of the table, the maximum time will be at most a constant period, $tS + t$).

Operation: Search for a record

Procedure: Compute the hashing address, and look in the blocks starting from the address. If all blocks of this bucket are full, look into the blocks for the next bucket until an empty block is encountered [WHY ?]

Performance: Quite fast ! (Almost constant time)

Operation: Deletion of a record.

Procedure, Performance: Similar to that for Searching for a record.

All the above are techniques used by the DBMS's to physically store each of the tables in a Database. In all of the above, we assume that each table is arranged according to the value of a sorting key. However, in practical use, a record may be searched by the value of some other key (in our example of Fig 7, we may have sorted the data using SSN, but search for a record given the Lname). This is a very common requirement, and often it is essential to get a fast data operation on a table based on more than one searching attribute. DBMS's provide another mechanism to quickly search for records, at the cost of using some extra disk space: **INDEXES**.

4.4. Basics of indexes

The main idea of an index is similar to the physical index (for example, there is an index at the end of your text book, that allows you to quickly look for details of some topics). Relational DB's can have two different types of indexes; we shall study each type below.

4.4.1. Primary Indexes

A **primary index file** is an index that is constructed using the ***sorting attribute of the main file***. We again consider the example of Figure 7. Notice that each block of this table can only hold a few records. Figure 8

below shows how we can construct a primary index file for this main file. In the primary index file, we store the value of the ***sorting attribute of the main file*** in the ***first record in each block*** of the main file. Corresponding to each entry, we store the address of the block. Notice that the index file is much smaller (i.e. has very few blocks) compared to the main file.

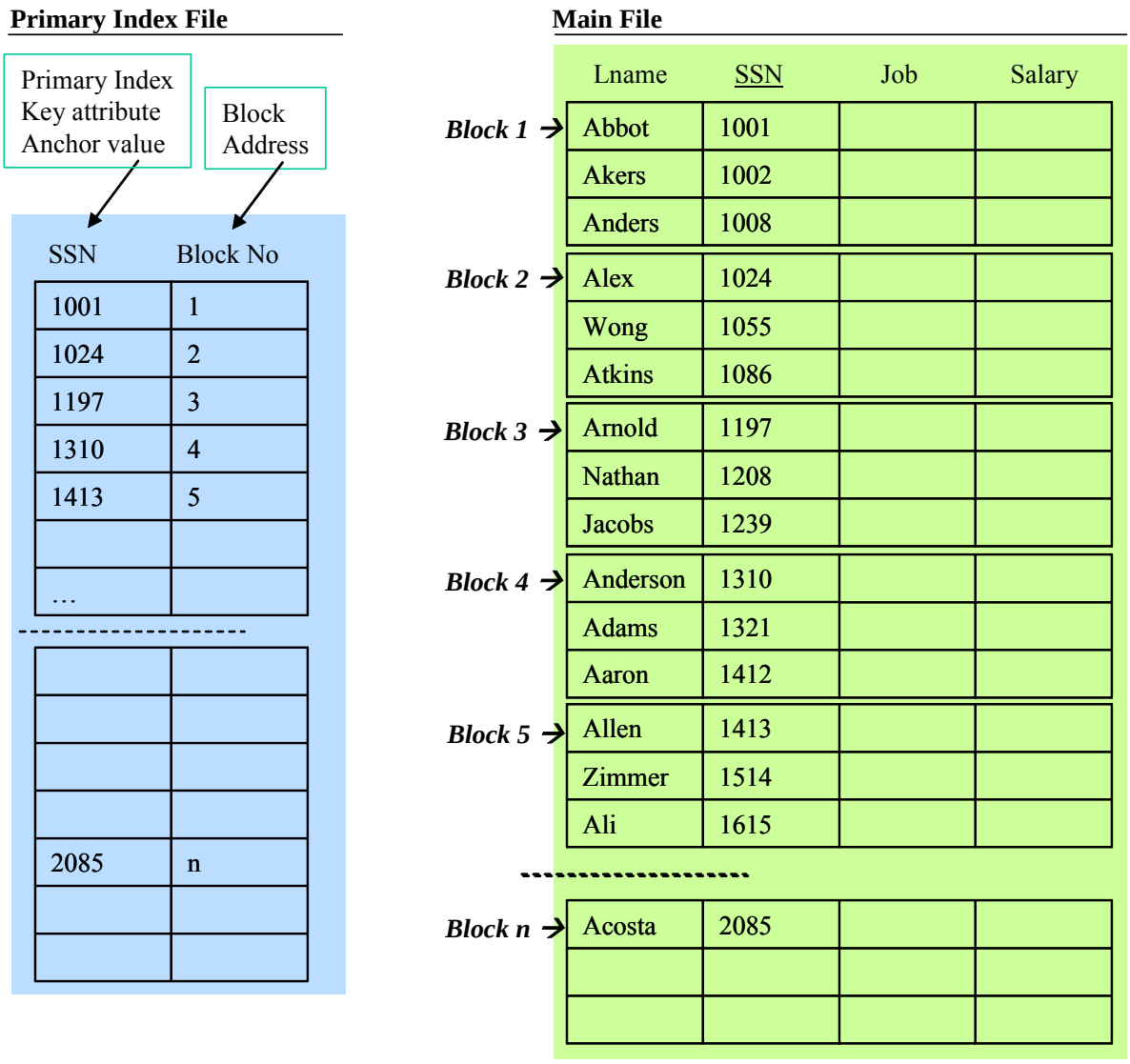

Figure 8. Main data file and its corresponding primary index file

Operation: Search for a record in the main file

Procedure: First perform a binary search on the primary index file, to find the address of the corresponding data. Suppose we are searching for 'SSN=1208'. From the primary index file we can tell that if a record for 'SSN=1208' exists, it must be in Block 3 [Why?]. We then fetch the contents of Block

3 from the HD, and search in this block for the record for 'SSN=1208'.

Performance: Very fast! The worst case search time to search for the address in the primary index file of size P blocks is ≈ $t(1 + \lg_2 P)$; an additional t units is needed to fetch the correct block from the main file. Thus the total time ≈ $t(2 + \lg_2 b)$

Similarly you can analyse the performance of other operations.

Problem: The Primary Index will work only if the main file is a sorted file. What if we want to insert a new record into the main file? Recall that inserting a record in a sorted file is time-consuming, therefore records are initially stored in an overflow file. Earlier we studied how simple heap files can be used for overflow files; in reality, most DBMS's will use a special structure in storing such records, called a ***balanced tree*** (or **B-tree**). However, in the following solution to our problem, we shall just assume that overflow files are heaps.

Solution: The new records are inserted into an unordered (heap) in the overflow file for the table. Periodically, the ordered and overflow tables are merged together; at this time, the main file is sorted again, and the Primary Index file is accordingly updated. Thus any search for a record first looks for the INDEX file, and searches for the record in the indicated Block. If the record is not found, then a linear search is conducted in the overflow file for a possible match.

NOTE: It is conventional for many DBMS's to use the primary key of a table as an ordering field, and therefore the primary index is conventionally constructed on the primary key.

4.4.2. Secondary Indexes

It is possible to construct indexes for any attribute of a table. Of course, since the main file can only be sorted by one field (usually the primary key field), therefore the contents of secondary indexes is different from that of primary indexes. We shall illustrate the concept of secondary indexes using the same example as Figure 7, on the attribute Lname. Note that our data file was sorted using SSN. Obviously, we cannot order the records of the main file using our secondary index key attribute, 'Lname' [Why?]. The **Secondary Index** is a two column file storing the ***block address*** of ***every secondary index attribute value*** of the table.

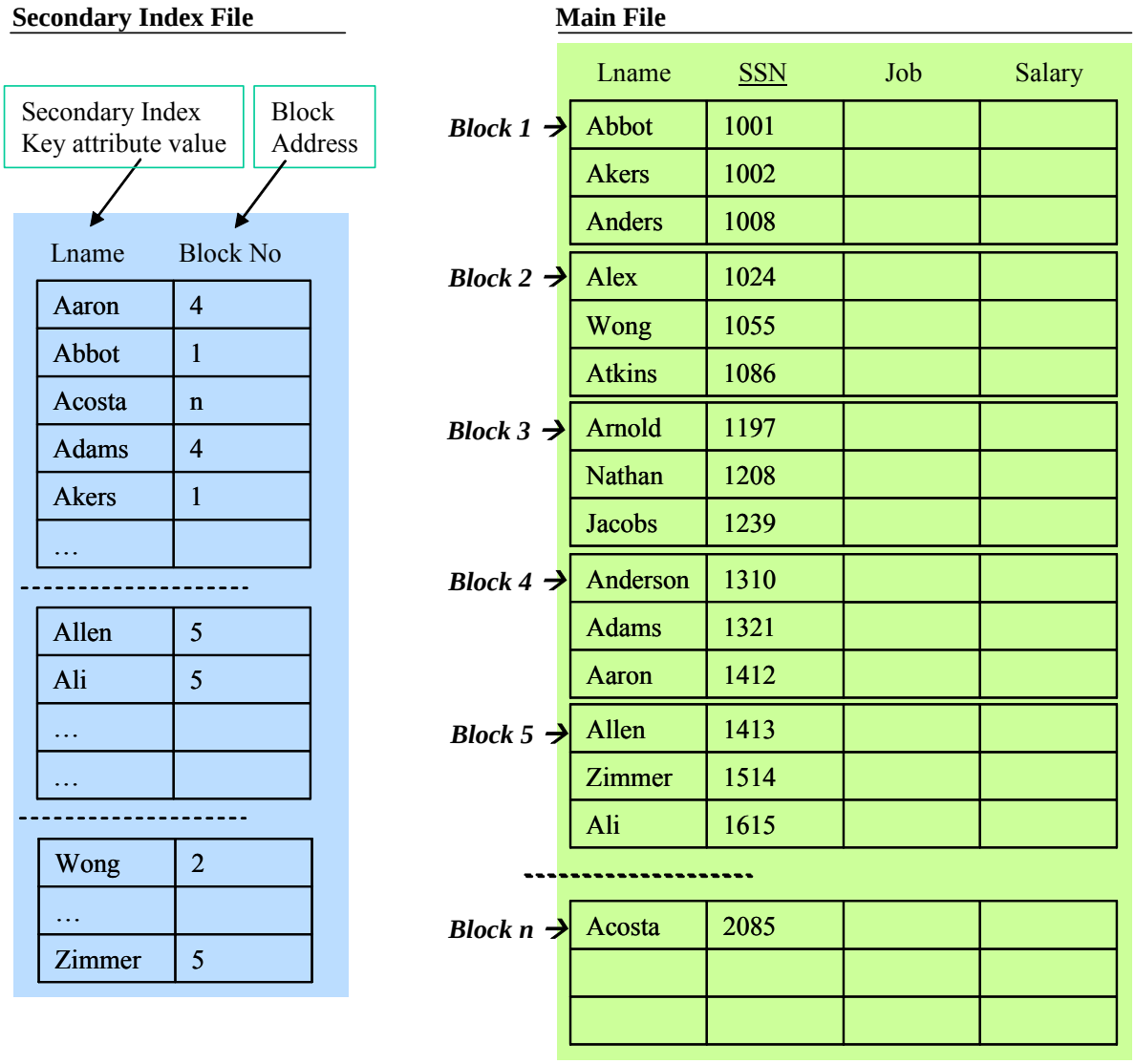

Figure 9. Schematic figure for a secondary index on *Lname* attribute of main file

You can also create Secondary Indexes for attributes that are non-unique. The idea is similar, though the storage details are slightly different (for example, if there are three persons with 'Lname=Wong', then the three records may be located in different blocks -- thus the secondary index file must record multiple entries for the same ordering attribute (Lname), one for each record -- and the search algorithm must account for each occurrence.

Of course, you can create as many secondary indexes as you wish. [Why would we like to create more than one Index for the Same Table?]

4.4.3. Creating Indexes using SQL

Example: Create an index file for Lname attribute of the EMPLOYEE Table of our Database.

CREATE INDEX myLnameIndex ON EMPLOYEE(Lname);

This command will create an index file which contains all entries corresponding to the rows of the EMPLOYEE table sorted by Lname in Ascending Order.

Example: You can also create an Index on a combination of attributes.

CREATE INDEX myNamesIndex ON EMPLOYEE(Lname, Fname);

4.4.4. DROP (delete) an index

Example: Delete the index created in the previous example.

DROP INDEX myNamesIndex;

In conclusion, indexes provide the advantage of speeding up access time for any of the DB's basic operations. The main disadvantages of Indexes are that they use up space on hard disk, and require some additional DB maintenance operations.