

DATABASES

(BASES DE DATOS)

2º CURSO F.I. I E.T.S.I.AP

UNIT 3 (3.5):

PHYSICAL DATABASE ORGANISATION

May 2009

CONTENTS

1. Introduction	3
2. The <i>File</i> Datatype	3
3. File Organisation and Access Methods	4
3.1 Disordered File (<i>Heap</i>)	5
3.2 Ordered File	5
3.3 Hash File (<i>Calculated addressing</i>)	6
3.4 Indexes	9
4. Relational Database Implementation	9

OTHER RECOMMENDED READINGS

Elmasri, R.; Navathe, S.

“Fundamentals of Database Systems”, Fourth Edition, Addison Wesley, 2003.

Date, C. J.

“An Introduction to Database Systems”, Eighth Edition, Addison Wesley, 2003.

1. INTRODUCTION

Database management systems (DBMS) introduce a new level of independence between users and data. This makes up a layer over the computer's operating system.

Given the large volume of information in databases and its expected existence during a period, the information must be stored in a secondary memory device, which nowadays generally implies the use of magnetic disks.

In this document, which corresponds to point 3.5 in the course, we review some basic ideas on data storage on disk, several file organisations, and the associated access methods which provide a means to obtain the stored information to be efficiently retrieved.

The data collection which conforms a database must be physically stored in some location. The DBMS will access this location to recover, modify and process these data, according the needs.

The databases generally store a great amount of data which must persist during long periods of time. This contrasts with the notion of programming datastructures which only need to last for a limited period of time. Most databases are permanently stored in magnetic disks, because of the following reason:

- Generally, databases are too large to fit in main memory.
- It is much more frequent that data is lost in main memory (volatile storage) than it happens with secondary storage (permanent storage).
- The storage cost for data unit is much lower for secondary storage than for main storage.

2. THE *FILE* DATATYPE

A file is a data structure for storage in secondary memory which consists of a collection of records. In many cases, all the records in the file are of the same type. This implies, on one hand, a waste of space in those cases in which not all the fields in the record have an assigned value, but, on the other hand, the programs which process files of this kind will be easier, since the start position and the size of each field are known and fixed.

In many systems, each relation in a relational database will be a database physical file, composed of records which store tuples. In other cases, the same relation is stored in several different files or two relations in the same file.

Since the accesses to secondary storage are much slower than the accesses to main memory, although not all accesses are equal, a good design of file structure must use the knowledge around the behaviour of disks in order to organise data in such a way that the access costs are minimised. A good design for a file structure will allow us to access all the information in a efficient way.

In what follows, we present some necessary concepts over files and their organisation, which will allow us to choose the appropriate physical structure for the storage of the database.

3. FILE ORGANISATION AND ACCESS METHODS

The *organisation* of a file refers to how data is organised in a file in terms of records, blocks and access structures. This includes the way in which records and blocks are stored in secondary memory and are linked between each other. The choice of a particular organisation depends on many factors, including the facilities to handle files and the use which the file will have. An *access method*, on the other hand, consists on a group of programs which allows for the application of operations of retrieval and update.

Those files whose organisation allows the finding of the exact location of the required record are called *direct access*. While a sequential search is an operation in $O(n)$, the direct access is in $O(1)$. This direct access can be obtained by different methods, depending on the file organisation, as we will see in the following sections.

In general, it is possible to apply several access methods to a same file organisation, although some other methods, though, can only be applied to files which are organised in a determined way.

The direct access to a record can be obtained through the relative record number (*RRN*) which is the relative position of a record since the beginning of a file. The *RRN* of the first record is 0, the second one is 1, etc. In these files, a correlative number is associated to each record which is being introduced in the file. If the file is composed of records of fixed and known length, we can use the *RRN* to compute the offset of a record with respect to the beginning of the file.

The access of records through their position does not help to find them based on a search condition. Nonetheless, it eases the construction of secondary access paths (indexes) to the file which will be studied in the following section.

An organisation, to have success, must allow to efficiently perform the operations which are supposed to be more frequent on the data. Logically, in many cases there is no organisation which allows an optimal solution to all the possible operations, so the selection

is done to give better solutions to more frequent operations. In other words, a trade-off must be found which takes this set of operations into account.

The three main primary file organisations are the disordered file or *heap*, the sequential file (ordered by a field) and the hash file (calculated addressing according to a field). Additionally, in some occasions, some auxiliary access structures, called *indexes*, are used to increase the speed of record retrieval in front of given search conditions.

3.1 Disordered File (*Heap*)

This is the simplest organisation. Records are stored in the file in the order in which are inserted, i.e., new records are inserted at the end of the file. This organisation is frequently used in association with other auxiliary access structures, such as indexes.

As can be easily deduced, the insertion of new records in a file of this kind is very efficient: the last block of the file is copied to a buffer, a new record is added after the last existing record in the block, and, finally, the block is overwritten on the disk. Looking for the first or the last record is relatively easy, since the address of the last block in the file is stored in the header of the file. Nonetheless, the search of a record based on some condition requires a sequential scan of the file until the record is found, or until the end of the file, if the record is not in the file.

The deletion of a record first involves a search, and then to copy the block where the record has been found into a memory buffer, to eliminate the record (releasing free space, or indicating it has been deleted with a flag) and to rewrite the block to disk.

3.2 Ordered File

When records are stored in a file according to the value of one of its fields (known as the ordering field), we obtain an ordered file (also known as sequential file). The advantages we get from this ordering are:

- The read of records in the file in the order defined by the ordering field is extremely efficient.
- Find the next record in the order of the ordering field does not require additional accesses to other blocks, because it is usually in the same block (unless the current record is the last in the block).
- The use of a search condition based on the ordering field allows a quick access if a binary search is used over the blocks.

With regards to the disadvantages which we can find in this organisation, we have the following:

- The ordering does not provide any advantage for the random or ordered access of records through a field which is not the ordering field. In order to have the same advantage for several fields we should have several copies of the file, each one ordered by a different field.
- The insertion and deletion of records are slow operations, since records must be remained ordered physically. In order to add a new record, first the place where it should be inserted have to be found according to the ordering field and, secondly, a new gap must be created for it. This second operation involves a great amount of time, since, on average, half of the records must be moved in each insertion. One option to make this insertion more efficient is to leave some free space in each block. Another method which is also used frequently consists in maintaining a temporary disordered file, known as overflow or transaction file, where new files are inserted. Periodically, the overflow file is ordered and fused with the ordered file (the main or master file). For deletion, the problem is not so important, because some records can be marked as deleted (with a flag) and a periodical reorganisation can recover this space.

The modification of the value of a field in an ordered file depends on two factors: the search condition to locate the record and the field to modify. If the search condition depends on the ordering field, a binary search can be done. Otherwise, a linear (sequential) must be performed. If the modification is over a field which is not the ordering field, the modification will only consist in changing the field in the record and rewriting the block. If the modification affects the ordering field, the record may change its position in the file, so a deletion of the old record and the insertion of the new one is needed. Finally, in order to read records in the order of the ordering field, first the file must be reorganised, sorting the overflow file and merging it with the main file, as well as the records which are marked as deleted with be effectively deleted. After all this, the file can be read sequentially.

Ordered files are not very common in current databases, unless an auxiliary access structure is used jointly with them: a primary index. This joint structure has the name of *indexed sequential files*.

3.3 Hash File (*Calculated addressing*)

This organisation provides a very quick access to records for some given search conditions. This search condition must be an equality condition over one field (known as *hash field*), which generally is a *key* for the file (attribute with uniqueness constraint). The idea of a hash field is to provide a function p known as *hash function* or randomisation function, which is applied to the value of the hash field of a record, and returns the address of the disk block

where the record is (or must be) located. The search of a record inside the block can be performed in a main memory buffer. For most records, only one block is needed.

The address space which is assigned to a file is composed of *buckets*, each one having many records. A bucket is, either a disk block or a group of contiguous blocks. Assume that we have M buckets, whose relative addresses go from 0 and $M-1$. We have to find a function which transforms the field value into an integer between 0 and $M-1$. A typical hash function is $p(K) = K \bmod M$, which returns the module of the quotient between K (the hash value) and M ; this value is the one used as relative address. A table which is maintained at the header of the file converts this bucket number into the corresponding address of the disk block. The values of the hash fields which are not integers can be converted to integer before the application of the function. For strings, we can associate codes to each character and use the product of all the character codes in a string.

Another hash technique consists in choosing several digits from the hash field (for instance, the third, the fifth and the eighth) in order to conform the hash address.

The problem which hash functions show is that they cannot guarantee that for different values we will get different addresses, because the number of different values the hash field can take is much greater than the number of addresses which are available for the records.

When the hash field value for a new record to be inserted corresponds to an address which already contains the maximum number of records which can fit a pocket, the process to find a solution to this problem is called *collision resolution*. There are several methods, the following amongst them:

- Open addressing: From the position which was occupied, the program seeks over the following positions, until it finds one where there is space.
- Chaining: In each pocket there is a pointer to a list of overflow records for that bucket.
- Multiple hashing: the program applies a second hash function if the first one produces a collision.

Each collision resolution method requires its own methods for insertion, retrieval and deletion of records. The goal of a good hash function is to distribute the records uniformly in the address space to minimise collisions without leaving too many unused buckets.

Main buckets

Overflow buckets

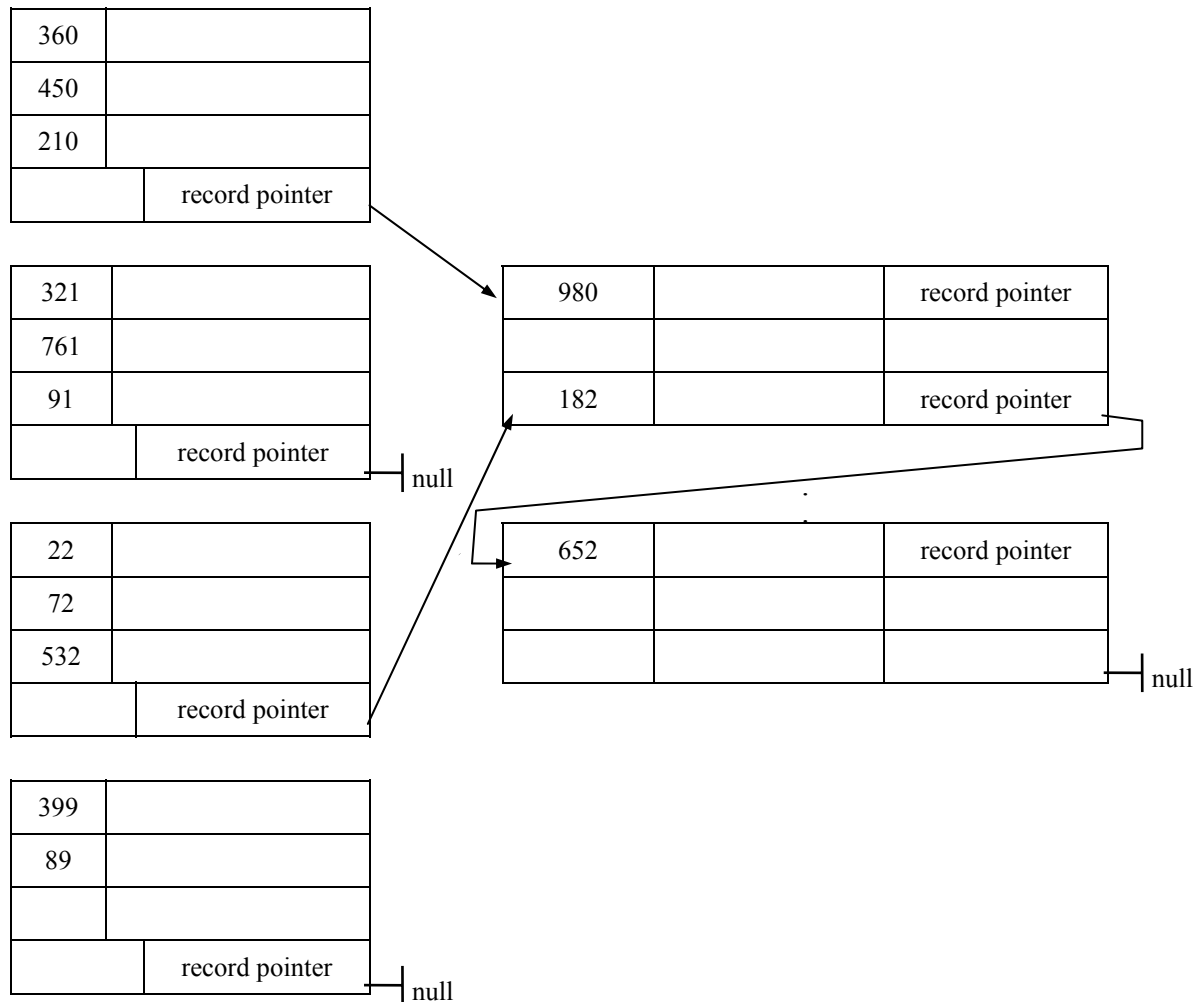


Figure.1 Dealing with overflow with chained buckets.

Another disadvantage of hash file is that the space which is reserved for the file is fixed. If we reserve M buckets and m is the maximum number of records which fit into a bucket, then in the file there will be room for $m * M$ records at most. If the number of records is much lower, then there is too much unused space. On the other hand, if the number of records increases considerably, there will be too many collisions and data retrieval will be slower because of the long overflow record lists. These problems can be solved by using techniques which use the dynamic expansion of the file (dynamic hashing).

Finally, with the hash method, the search of a record through the value of a field which is not the hash field is equally costly than with disordered files.

3.4 Indexes

Assuming that there is already a file with a primary organisation as the previously described, we can define auxiliary access structures called *indexes*, which are used to increase the speed of record retrieval in response to certain search conditions.

Indexes provide secondary access methods to access records in an efficient way without affecting its physical position inside the main file. Any field in a file can be used to create an index. It is then possible to have many indexes for a single file.

Some kinds of indexes can only be constructed over files with a determined primary organisation.

An index is an access structure which is usually defined over a single field in a file, called the *index file* (although it is also possible to construct it over a set of fields). All indexes have the same basic type of record, composed of a pair of elements: the search field and address field. An index makes it possible to impose an order in the file without physically ordering it, which makes much easier to insert record than it is with ordered files.

It is important to distinguish between two kinds of indexes:

- **Primary index.** It is an ordered file whose records are of fixed length and are composed by two fields. The first field is of the same type of the key field in the main ordered file and the second is an address to the disk block. The ordering field is then called primary key. This index is defined over data files which are ordered by the ordering field.
- **Secondary index.** This is also a file whose records have two fields: the first field is of the same type of some field in the main file (which is not an ordering field), the second field is a pointer to a block or to a record. There can be many secondary indexes for a file. For instance, assume that we want to build a secondary index over a key field (without repeated values). We will call this field secondary key. In this case, there will be an entry in the index for each data record, which will contain a value for the secondary key and a pointer to a block which contains the record with that value, or a direct pointer to the record.

4. RELATIONAL DATABASE IMPLEMENTATION

Each DBMS offers a great variety of options to organise files and access paths. This generally includes many types of indexes, the possibility of clustering disk block with related records, links through pointer between related records, and different kinds of hashing. When a physical database design is performed, several options must be taken among all the options

provided by the DBMS, having several factors into account: response time, space required, access structures, execution frequencies for queries and transactions, and other requirements which are specified for the database. The good behaviour will also depend on the size of the record and the number of records in the file. The attributes which are expected to be used frequently to recover records must have paths for primary access or secondary indexes. Sometimes, because of the requirements of the database system, it is necessary to reorganise some files by constructing new indexes or changing the primary access methods.

A very popular option to organise a file in a relational database is to store all the tables as disordered files and to create as many secondary index as needed. The attributes which are frequently used for selects and joins are candidates for these indexes. If the records are going to be retrieved very frequently in the order of one attribute, the option of a primary index over an ordered file must be considered. If the file is going to suffer many insertions and deletions, the number of indexes must be minimised.

In many systems, the index is not an integral part of the file, but it is created or destroyed dynamically. When the administrator thinks that a file is going to be accessed very frequently through a condition which implies a given field, s/he configures the DBMS in such a way that an index is created over that field, an index which will be generally secondary in order to avoid a physical ordering. Only in some systems users/administrators can also create primary indexes. A file which has a secondary file over each of their fields is called a *completely inverted file*.

If a file is not going to be used commonly to recover records in a given order, we can use a hash file. The hash organisation must be dynamic if the size of the file changes frequently.

We have already seen how queries can be optimised through indexes. Nonetheless, a trade-off must be found between the efficiency for searches and for insertions/deletions, before determining how many indexes must be created.

For the case where queries are performed to more than one relation, there are other techniques. This is the case of relation *clusters*. When two relations have two attributes through which it is common to make joins, many times it is preferable to physically store both tuples in the same block. For instance, assume that in relation $R2$ we have a foreign key defined which refers to the primary key of relation $R1$, as shown in figure 2:

R1 (a1 : dom1, a2 : dom2) PC: {a1}		R2 (b1 : dom3, b2 : dom4, b3 : dom1) PC: {b1} FK: {b3} → R1 (a1)		
a1	a2	b1	b2	b3
12	Twelve	9A	ASDF	84
51	Fifty-one	0B	QWER	51
84	Eighty-four	1L	ZXCV	12
		2X	QAZ	12
		3P	POIU	84
		4K	MNBV	51
		5T	TTTT	51
		6M	MMM	12

Figure 2: Two relations which are linked by a foreign key

If the join between $R1$ and $R2$ through attributes $a1$ and $b3$ is common, it is advisable to put both relations in a cluster. The cluster is a storage method which allows the store of a tuple of $R1$ with all the tuples of $R2$ which have in its foreign key the same value that the primary key of the other relation. This value is known as the *cluster key*.

By default, all tuples which are related by the same value for the cluster key are stored in the same block, and in a different block for each different value. If all the tuples do not fit into the block, chaining is used to solve this overflow.

BLOCK 1		BLOCK 2		BLOCK 3	
a1	a2	a1	a2	a1	a2
12	Twelve	51	Fifty-one	84	Eighty-four
	b1		b1		b2
	1L		0B		9A
	2X		4K		3P
	6M		5T		MMM
					ASDF
					POIU

Figure 3: Storing relations $R1$ and $R2$ in a cluster.

When storing related tuples from two different tables in the same block, the benefit is double:

1. The access time to make joins is reduced, because the number of blocks which is access is lower.
2. The value for the cluster key is only stored once, which involves a saving in space, because foreign keys do not need to be stored.

The cluster of two or more relations, as we have seen, puts together the tuples with the same value in the cluster key. This entails a reduction in the efficiency of performance of operations such as insertions and modifications. Consequently, the decision to make a cluster must take this fact into account.