FILE ORGANIZATIONS AND INDEXES

1. INTRODUCTION

A **file organization** is a way of arranging the records in a file when the file is stored on disk. A file of records is likely to be accessed and modified in a variety of ways, and different ways of arranging the records enable different operations over the file to be carried out efficiently.

A DBMS supports several file organization techniques, and an important task of a DBA is to choose a good organization for each file, based on its expected pattern of use.
Each file organization makes certain operations efficient, but often we are interested in supporting more than one operation.

2. COST MODEL

Notation and assumptions:

- there are *B* data pages with *R* records per page,
- the average time to read or write a disk page is *D*,
- the average time to process a record (e.g., to compare a field value to a selection constant) is **C**.
- In the hashed file organization, we will use a function, called a *hash function*, to map records into a range of numbers; the time required to apply the hash function to a record is *H*.

Typical values today are *D* = 20 milliseconds, *C* and *H* = 1 to 10 microseconds; we therefore expect the cost of I/O to dominate.

3. COMPARISON OF THREE FILE ORGANISATIONS

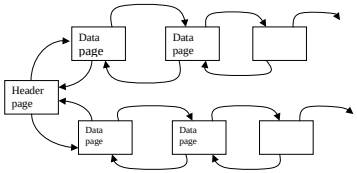
We compare the costs of some operations for three basic file organizations:

- ***heap files****;*
- ***files sorted on some field****;*
- ***files that are hashed on some field****.*

The operations that we consider are the following:

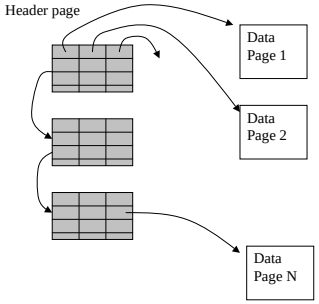
- **Scan**: Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. There is also a CPU overhead per records for locating the record on the page (in the pool).
- **Search with Equality Selection**: Fetch all records that satisfy an equality selection, e.g., „Find the Students records for the student with `sid` 23." Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.
- **Search with Range Selection**: Fetch all records that satisfy a range selection, e.g., „Find all Students records with `name` alphabetically after 'Smith'."
- **Insert**: Insert a given record into the file. We must identify the page in the file into which the new records must be inserted, fetch that page from disk, modify it to include the new record, and the write back the modified page. Depending on the file organization, we may have to fetch, modify and write back other pages as well.
- **Delete**: Delete a record that is specified using its `rowid`. We must identify the page that contains the record, fetch if from disk, modify it, and write it back. Depending on the file organization, we may have to fetch, modify and write back other pages as well.

3.1. Heap Files



Heap file – doubly linked list of pages:

- linked list of pages with free space
- linked list of full pages



Heap file organization with a directory

Free space is managed by maintaining a bit per entry indicating whether the corresponding page has any free space on the page.

Scan: The cost is $B(D + RC)$ because we must retrieve each of B pages taking time D per page, and for each page, process R records taking time C per records.

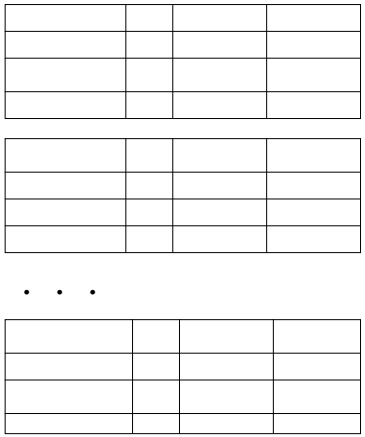
Search with Equality Selection: The cost is $0.5B(D + RC)$. If there is no record that satisfies the selection we must scan the entire file to verify this.

Search with Range Selection: The cost is $B(D + RC)$.

Insert: The cost is $2D + C$.

Delete: The cost is the cost of searching plus $C + D$.

3.2 Sorted Files



Scan: The cost is $B(D + RC)$ because all pages must be examined.

Search with Equality Selection: The cost is $Dlog_2B + Clog_2R$, which is a significant improvement over searching heap files (the so called - *binary search*)

Search with Range Selection: The cost is the cost of search plus the cost of retrieving the set of records that satisfy the search. The cost of the search includes the cost of fetching the first page containing qualifying, or matching records.

Insert: The cost is the cost of searching to find the position of the new record plus 2 * $(0.5B(D + RC))$, that is, search cost plus $B(D + RC)$.
Remarke: use overflow or transaction file – unordered file

Delete: The cost is the same as for an insert, that is, search cost plus $B(D + RC)$.

3.3. Hashed Files

A hashed file has an associated search key (called the **hash key**), which is a combination of one or more fields of the file. The idea behind hashing is to provide a function h, called a hash function that is applied to the hash field value of a record and yields the address of the disk block in which the record is stored.

3.3.1. Internal hashing

Given an array of records with M slots. The index array range from 0 – M-1.

	Name	SSN		Salary
0				
1	Morzy			
2				
3				
M-2				
M-1				

Hash function transforms the hash field value into an integer between 0 and M-1. Common function:

H(K) = K mod M

The whole process consists of two steps:

- folding – transformation of hash field values into integers
- hashing

Collision – when the hash field value of a new record that is being inserted hashes to an address that already contains a different record.

Collision resolution – the process of finding another position for the inserted record.

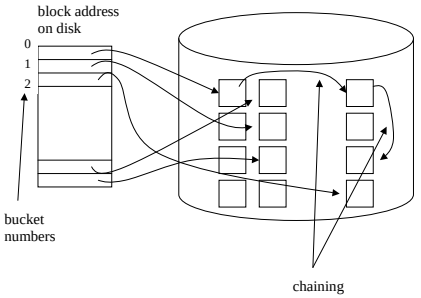
Collision resolution methods:

- **open addressing** – check the subsequent position,
- **chaining** – use overflow locations,
- **multiple hashing** – apply a second hash function.

Each collision resolution method requires its own algorithms for insertion, retrieval and deletion of records.

3.3.2. External hashing

Hashing for disk files is called **external hashing**. The pages in a hashed file are grouped into **buckets**. The hashing function maps a hash key into a relative bucket number. A table maintained in the file header converts the bucket number into the corresponding disk block address.



Scan: The number of pages and the cost of scanning all the data pages is about 1.25 times the cost of scanning an unordered file that is, 1.25$B(D + RC)$.

Search with Equality Selection: The cost is $H + D + RC$.

Search with Range Selection: The cost is 1.25$B(D + RC)$.

Insert: The cost is the cost of search plus $C + D$.

Delete: The cost is again the cost of the search plus $C + D$ (for writing the modified page).

3.4. Choosing File Organization

Figure 1 compares I/O costs for the three file organizations.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap	BD	$0.5\,BD$	BD	$2D$	*Search* + D
Sorted	BD	$Dlog_2B$	$Dlog_2B$ + # *matches*	*Search* + BD	*Search* + BD
Hashed	$1.25BD$	D	$1.25\,BD$	$2D$	*Search* + D

Figure 1. A Comparison of I/O Costs

4. OVERVIEW OF INDEXES

- An index on a file is an auxiliary structure designed to speed up operations that are not efficiently supported by the basic file organization of records
- An index can be viewed as collection of **data entries**, with an efficient way to locate all data entries with search key value k. Each such data entry, denoted as k^*, contains enough information to retrieve (one or more) data records with search key value k.

Two important questions to consider are:

- How are data entries organized in order to support efficient retrieval of data entries with a given search key value?
- Exactly what is stored as a data entry?

5. PROPERTIES OF INDEXES

5.1 Clustered versus Unclustered Indexes

When file is organized so that the ordering of data records is the same as or close to the ordering of data entries in the index – we say that the index is **clustered**. Otherwise, the order of the data records is random, defined purely by their physical order – **unclustered** index

In practice, data records are rarely maintained in full sorted order, unless data records are sorted in an index using Alternative (1), because of the high overhead of moving data records around to preserve the sort order as records are inserted and deleted.

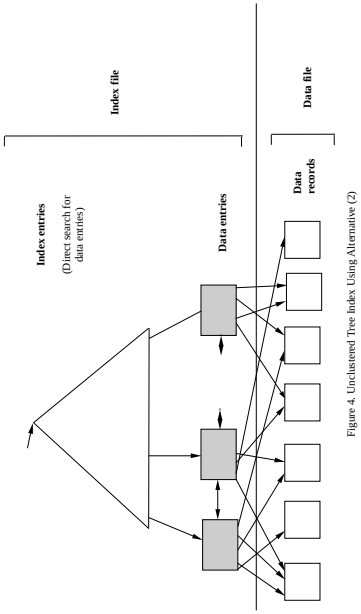


Figure 4. Unclustered Tree Index Using Alternative (2)

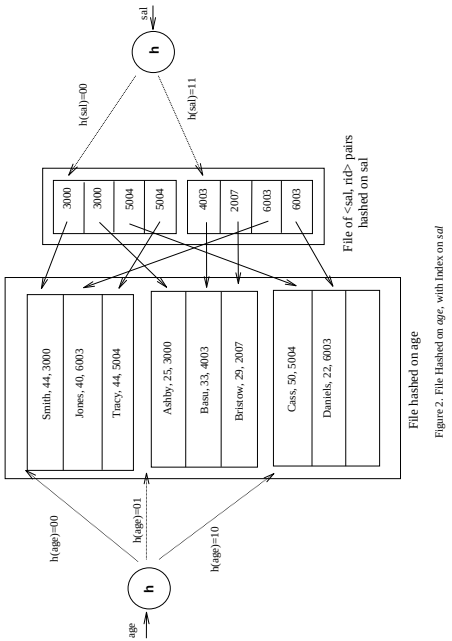


Figure 2. File Hashed on *age*, with Index on *sal*

Alternatives for Data Entries in an Index

A data entry *k** allows us to retrieve one or more data records with key value *k*. We need to consider four main alternatives:

1. A data entry *k** is an actual data record (with search key value *k*).

2. A data entry is a <*k*, *rowid*> pair, where *rowid* is the record id of data record with search key value *k*.

3. A data entry is a <*k*, *rowid-list*> pair, where *rowid-list* is a list of record ids of data record with search key value *k*.

4. A data entry is a <*k*, *bitmap*> pair, where *bitmap* is a list of 0 and 1 representing records.

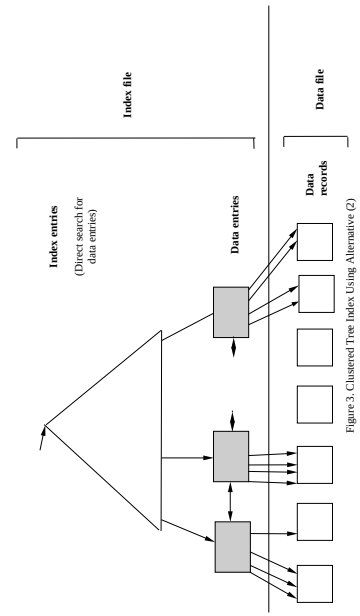


Figure 3. Clustered Tree Index Using Alternative (2)

Remarks:

- records are sorted initially and each page is left with some free space to absorb future insertions
- if the free space on a page is subsequently used up (by records inserted after the initial sorting step), further insertions to this page are handled using linked list of overflow pages
- after a while, the order of records only approximates the intended sorted order, and the file must be **reorganized** (i.e., sorted afresh) to ensure good performance.
- a data file can be clustered on at most one search key - we can have at most one clustered index on a data file
- we can have several unclustered indexes on a data file.

5.2 Dense versus Sparse Indexes

- An index is said to be **dense** if it contains one data entry for every search key value that appears in a record in the indexed file
- An index is said to be **non dense or sparse** if it contains one data entry for each page of records in the indexed file
- We cannot build a sparse index that is not clustered - thus we can have at most one sparse index. A sparse index is typically much smaller than a dense index.
- A data file is said to be **inverted** on a field if there is a dense secondary index on this field. A **fully inverted** file is one in which there is a dense secondary index on each field that does not appear in the primary key.

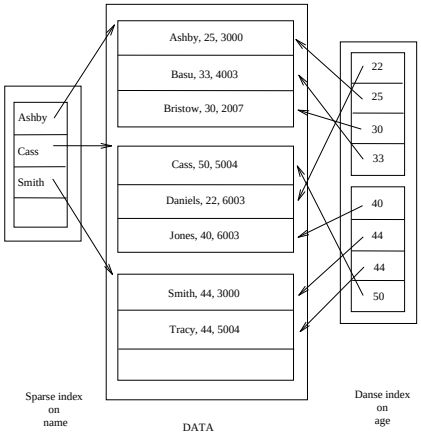


Figure 5. Sparse versus Dense Indexes

5.3 Primary and Secondary Index

An index on a set of fields that includes the *primary key* is called a **primary index**. An index that is not a primary index is called a **secondary** index. (The terms *primary index* and *secondary index* are sometimes used with a different meaning: An index that uses Alternative (1) is called a primary index, and one that uses Alternative (2) or (3) is called a secondary index.)

Two data entries are said to be **duplicates** if they have the same value for the search key field associated with the index. A primary index is of course guaranteed not to contain duplicates; a secondary index contains duplicates.

5.4 Indexes Using Composite Search Keys

The search key for an index can several fields; such keys are called **composite search keys** or **concatenated keys**. As an example, consider a collection of employee records, with fields *name*, *age*, and *sal*, sorted in sorted order by *name*. Figure 6 illustrates the difference between a composite index with key *<age, sal>*, a composite index with key *<sal, age>*, an index with key *age* and an index with key *sal*. All indexes shown in the figure use Alternative (2) for data entries.

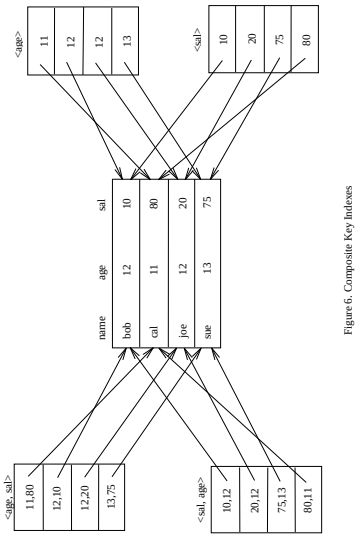


Figure 6. Composite Key Indexes

We distinguish two basic types of queries:

- an **equality query** is one in which *each* field in the search key is bound to a constant. For example, we can ask to retrieve all data entries with *age* = 20 and *sal* = 10.

- a **range query** is one in which not all fields in the search key are bound to constants. For example, we can ask to retrieve all data entries with *age* = 20; this query implies that any value is acceptable for the sal field. As another example of a range query, we can ask to retrieve all data entries with *age* < 30 and *sal* > 40.

6. INDEX SPECIFICATION IN SQL-92

The SQL-92 standard does *not* include any statement for creating or dropping index structures. The following command to create a B+ tree index is illustrative:

```
CREATE INDEX IndAgeRating ON Students
          WITH STRUCTURE =BTREE,
             KEY = (age, gpa)
```

This specifies that a B+ tree index is to be created on the Students table using the concatenation of the *age* and *gpa* columns as the key. Thus key values are pairs of the form <*age*, *gpa*>, and there is a distinct entry for each such pair. Once the index is created, it is automatically maintained by the DBMS adding/removing entries in response to inserts/deletes on Students.