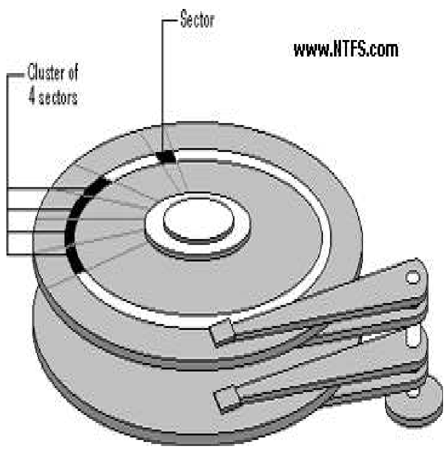
Principles of Databases

Storage

David Sinclair

Hard Disks

- Disk drive consists of a set of *Platters*.
- Platters are generally spinning (∼2400rpm).
- One head read/writes at a time.
- Each platter is divided into a set of *tracks*.
- The terms *page* and *block* are used interchangeably, but strictly
  - A *page* is the smallest unit supported by OS.
  - A *block* is a multiple of a *page* and is the smallest unit supported by an application.
- A *cylinder* is all the tracks reachable from one arm position (virtual).

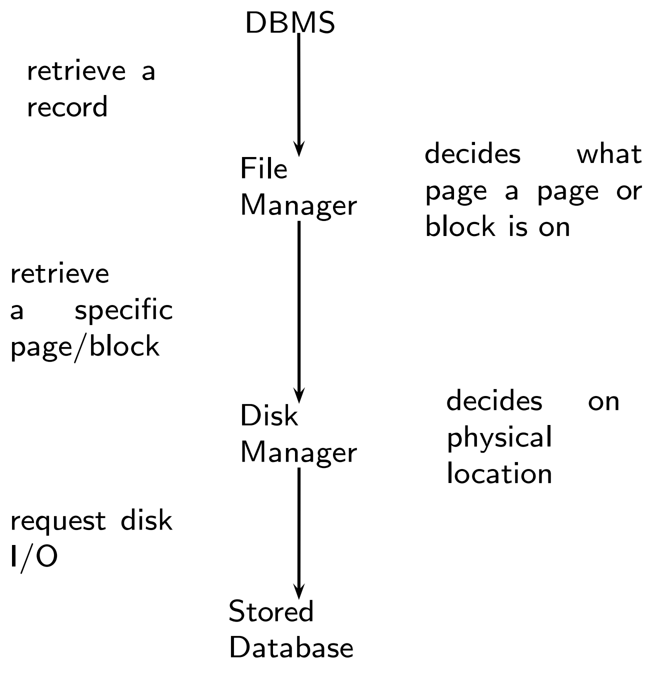


Hard Disk (2)

- The time to access a block:
  1. *Seek Time*: Time to move arm to position the head on the correct track (∼6ms).
  2. *Rotational Delay*: Time to wait until the correct block is under the read/write head (∼3-6ms).
  3. *Block Transfer Time*: Time to move data from/to disk surface (∼0.2ms per 8k page).
- Total time to locate and read/write block = 1+2+3.
- Since 1 & 2 >> 3, we need to minimise 1 & 2.
- We can do this by contiguous data on:
  - consecutive pages on the same track; then
  - consecutive tracks on the same cylinder; and then
  - on adjacent cylinders.
- $\textit{Bulk Transfer Time} = \frac{1}{\textit{transfer time of consecutive blocks}}$

Disk and File Managers

- *Page/Block* is the unit of transfer between disk and memory.


DBMS
retrieve a record
File Manager
decides what page a page or block is on
retrieve a specific page/block
Disk Manager
decides on physical location
request disk I/O
Stored Database

- *Disk Manager*: OS component managing free space on disk, performs garbage collection and defragmentation.
- *File Manager*: Associates file names with sets of blocks/pages; may be part of OS or DBMS if OS is not suitable.

Clustering on Disk Surfaces

- *Clustering*: Placing logically related records physically close together on disk.
- DBA can vary clustering in the mid-life of the database.
- Knowledge of how data is to be used is essential for good database design:
  - How frequently data is to be accessed determines storage mechanism.
  - What are the nature of the queries:
    - Exact: `SELECT * WHERE x=y`
    - Range: `SELECT * WHERE x < 100 AND y > 40`

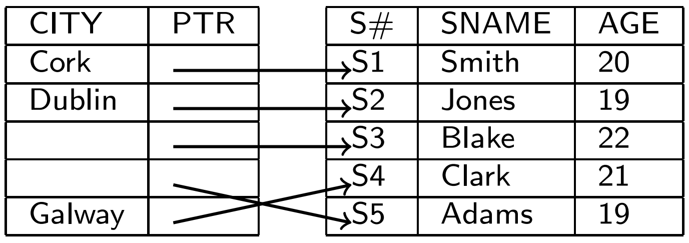
Index Files

- Large files can take a long time to search.
- Index files, e.g. the index at the back of a book, can speed this up.
- Can have multiple index files, e.g. a medical book can have an index for diseases and another for drugs.
- *Index*: a sorted list of records/rows residing in the table
- Two ways to execute a query:
  - Sequential search through (unsorted) file.
  - Binary Search through sorted index to find offset into data file

Sample Data

S#	SNAME	AGE	CITY
S1	Smith	20	Cork
S2	Jones	19	Dublin
S3	Blake	22	Dublin
S4	Clark	21	Galway
S5	Adams	19	Dublin

Indexed on CITY

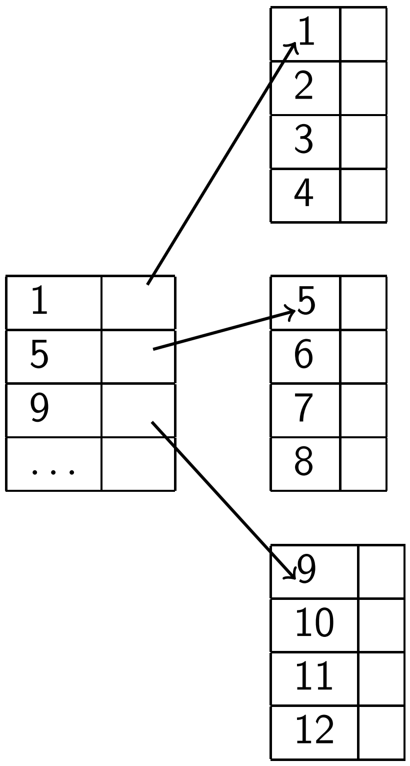


CITY	PTR
Cork	
Dublin	
Galway	

S#	SNAME	AGE
S1	Smith	20
S2	Jones	19
S3	Blake	22
S4	Clark	21
S5	Adams	19

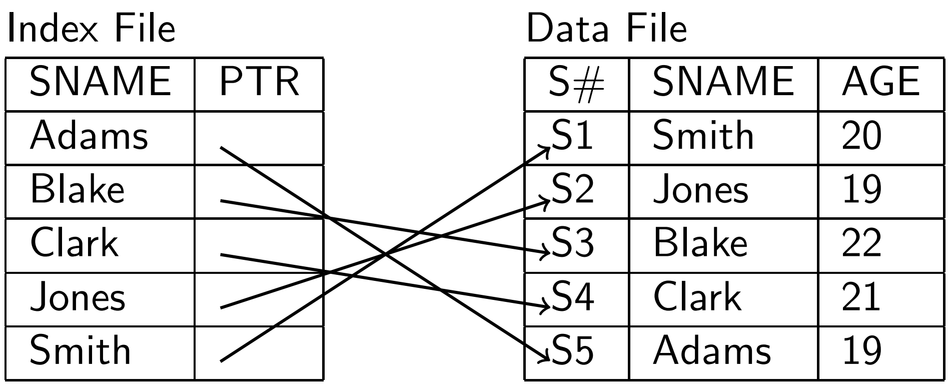
Index Files (2)

- Ordering Field
  - Physically orders records of files on the disk based on one of their fields.
  - If the *ordering field* uniquely identifies each row in the table then it called either a *Primary Key* or a *key field* or a *ordering key*.
- Primary Index
  - Ordered file with fixed length records of two fields:
    1. Same datatype as the ordering key field of the data file.
    2. Pointer to a disk block



Index File (3)

- Secondary Index:
  - As with Primary Index, it is an ordered file with two fields:
    1. Indexing Field: Same datatype as data file's non-ordering field.
    2. Block Pointer or Record Pointer



Index File

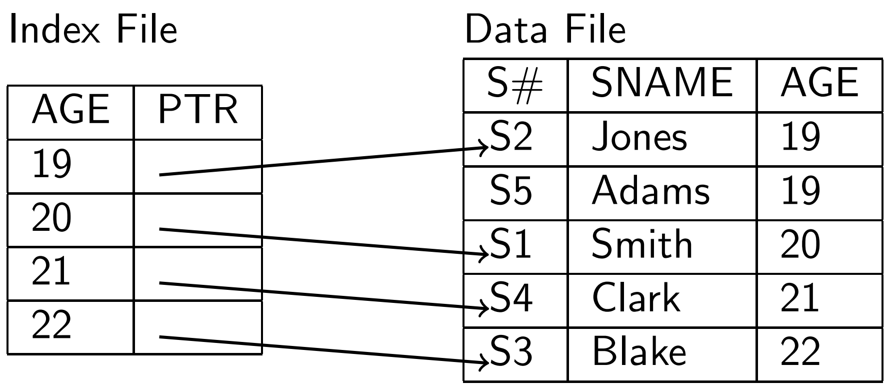
SNAME	PTR
Adams	
Blake	
Clark	
Jones	
Smith	

Data File

S#	SNAME	AGE
S1	Smith	20
S2	Jones	19
S3	Blake	22
S4	Clark	21
S5	Adams	19

Exotic Indexing

- Clustering Index:
  - Determines physical ordering of row based on a non-key field without distinct values for each record.
  - Only one per table but each can comprise multiple columns, e.g. phone book organised by family name and given name.



Index File

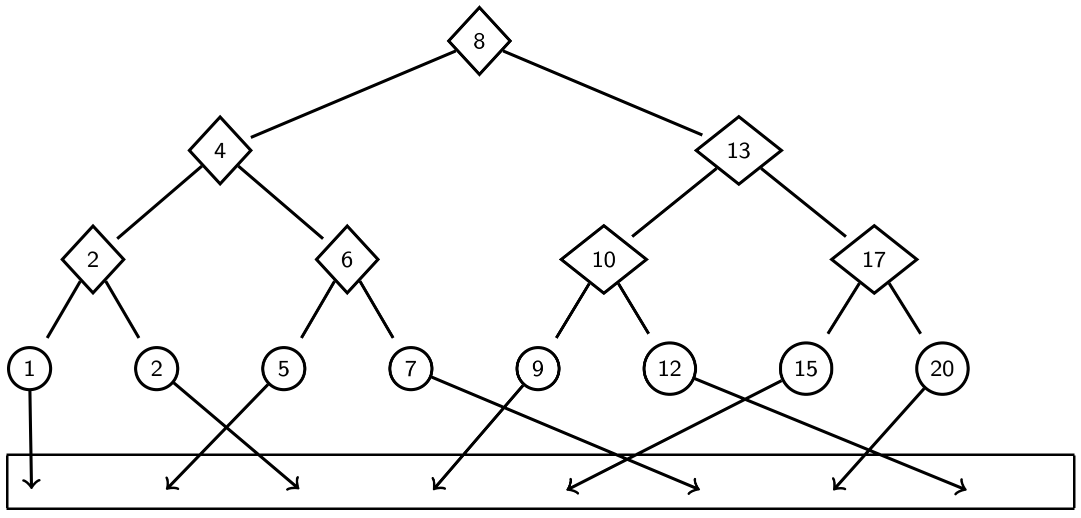
AGE	PTR
19	
20	
21	
22	

Data File

S#	SNAME	AGE
S2	Jones	19
S5	Adams	19
S1	Smith	20
S4	Clark	21
S3	Blake	22

Exotic Indexing (2)

- Binary Tree Index:
  - It compares the values to be searched with the current node. If the value is less than the current node it goes to the left child node, otherwise it goes to the right child node.
  - Continues until it reaches a leaf node. If the leaf node contains the target value, it retrieves the index into the data file.



Indexing

- A file can have any number of indexes.
- Can create an index on a primary key or on another field or combination of fields.
- An index on a field combination is not the same as two separate indexes.
- Indexes usually speed up retrieval but slow down updates.
- Binary tree index is usually the best all-round index.

Hash Indexing

- Also known as *Hash Addressing*.
- Index is a pointer to a storage location.
- *Hash Addressing* “randomises” the storage location of a record. Given a record, the *hash function* calculates a location to store the record based on the record’s contents.
- Ideally this calculated storage location (*hash value*) is *unique*.
  - Ideally only one record should be mapped to a given storage location.
  - When more than one record is mapped to the same location we have a *hash collision*.
- DBMS calculates the hash address (*hash value*) and tells the file manager to store the record at this location.
  - Sometimes a cluster of blocks is used in order to store multiple records with the same hash value.

Hash Indexing (2)

Student ID (k)	Location, $H(k) = k\ mod\ m$, $(m = 13)$
100	100 mod 13 = 9
200	5
300	1
400	10
500	6

0	1	2	3	4	5	6	7	8	9	10	11	12
	300				200	500			100	400		

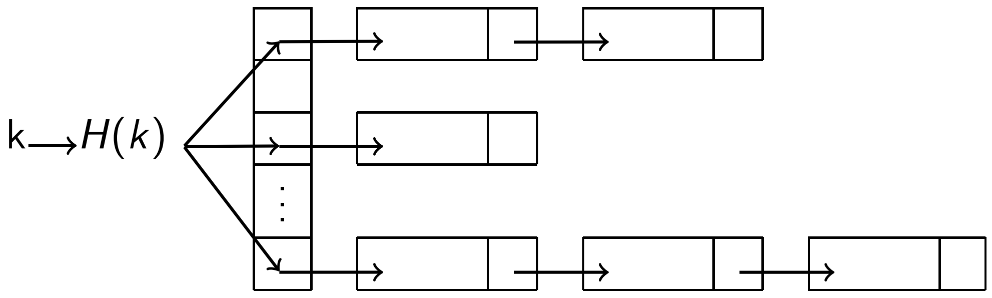
Hash Collisions

Student ID (k)	Location, $H(k) = k\ mod\ m$, $(m = 13)$
400	10
700	11
1000	12
1200	4
1700	10

0	1	2	3	4	5	6	7	8	9	10	11	12
				1200						X	700	1000

- Range of values is greater that number of locations $\Rightarrow$ *collisions*
- Generally Hash Tables should be about 50% to 70% full to reduce probability of collisions and not to waste space.

How to deal with Hash Collisions



- Hash addresses point to entries in a *Bucket Directory*.
- The *Bucket Directory* points to the *Primary Bucket* that stores the first value hashed to the corresponding *Bucket Directory* entry.
- Collisions are handles by the *Primary Bucket* pointing the a chain of *Overflow Buckets*.
- This collision resolution technique is called *chaining*.
- Hash function should scatter values evenly across the *Bucket Directory* to avoid having long chains for a few buckets.

How to deal with Hash Collisions (2)

- *Open Addressing*: If a collision occurs, the next probes (search for an empty hash address) are performed according to the formula $h_i(k) = (hash(k) + f(i))\ mod\ m$ where:
  - $h_i(k)$ is an index in the hash table to insert k
  - $hash(k)$ is the hash function
  - $f(i)$ is the collision resolution function
  - i is the number of the current attempt to insert k
- $f(i) = i$: *linear probing*
  - Insert: If collision, probe next slot
    - If unoccupied, store k there.
    - If occupied, continue probing next slot
  - Disadvantage is that large cluster tend to form.
- $f(i) = i^2$: *quadratic probing*
  - Insert: If collision, probe next slot, if occupied, continue probing slot $i + 4$, then $i + 9$ etc.
- *Multiple Hashing*: Use a 2^{nd} hash function for the collision resolution function.

Dynamic Hashing

- Up to now we have been considering fixed size hash tables, *static hashing*.
- If the underlying data table grows, then the overflow chains get larger and spoil the predictable hash table performance.
- If the underlying data tables shrink significantly, then the hash table may be wasting space with many unallocated slots.
- As a rule of thumb, the hash table should be 125% of the expected data capacity to yield good performance.
- Worst case scenario: hash table degrades in a linear list with one long chain of overflow buckets.
- *Dynamic Hashing* schemes have been developed to overcome this problem by adapting the hash function and allowing hash tables to grow and shrink.

Dynamic Hashing (2)

- Extendible Hashing:
  - A type of dynamic hashing where keys are grouped as per the first i bits in their code.
  - Each group is stored in one block.
  - When a block is full split the group in 2 and use $i + 1$ bits to find the record's location.
  - Example:
    - Each disk block can contain 3 records.
    - 4 groups of keys differentiated by their first 2 bits, i.e. 00, 01, 10 and 11.

00	01	10	11
00010	01001	10001	11000
00100	01010	10100	11010
	01100		

Dynamic Hashing (3)

- Adding 11 (=01011) to the hash table yields:

000/001 still on same block	010	011	100/101 still on same block	110/111 still on same block
00010	01001	01100	10001	11000
00100	01010		10100	11010

- Other Hashing Techniques
  - Linear Hashing is another type of dynamic hashing where the table grows by one slot at a time and uses a family of hash functions.

Disadvantages of Hashing

- Stored files can have any number of indexes but only one hash table.
- Works well for single equality predicates only.
- Physical sequence on disk does not correspond to any logical organisation leading to high seek times and thrashing (where OS spends a lot of time repeatedly moving data from the hard disk and (virtual) memory).
- As file size increases, the number of collisions increases.
- Ideal hash function would be uniform and random.
- Hard to come up with a good hash function.
  - Birthday Paradox: There are 365 possible birthdays but is only takes 24 people for the probability of two of them having the same birthday to be greater than 0.5.

Different File Organisations

Operation	Unsorted	Sorted (i.e.Index)	Hash
Search	-(linear search)	=(binary search)	+(apply hash function)
Insert	+(insert at end)	-(reorganise file)	=(collisions might occur)
Delete	-(find it first)	-(reorganise file)	+(easy to find, no need to reorganise file)
Modify	=(difficult to find, easy to modify)	-(reorganise if sorting field is affected)	-(reorganise if hash field is affected)