



COSC210 Database Management Systems

Lecture 9 - Disk Storage, File Structures, Hashing and Indexing

Dr. Mitchell Welch / Dr. Edmund Sadgrove

Reading

- Chapter 16, 17 from *Fundamentals of Database Systems* by Elmazri and Navathe

Summary

- Hardware Overview
- File Records and Operations
- Files of Ordered/Unordered Records
- Hashing Techniques
- Single Level Indexes
- Multilevel Indexes

Hardware Overview

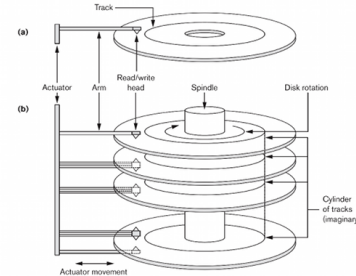
- Up until now we have covered:
 - The **Database Schema**.
 - The Three-Schema Architecture:
 - The **External Schema**. ✓
 - The **Conceptual Schema**. ✓
 - The **Internal Schema**.
- In this lecture we will look at the **Internal Schema**.
- This topic is fairly high-level as much of the content will be covered in detail in other units.

Hardware Overview

- Data is stored in bytes with 8 (binary) bits.
- Memory has three main types:
 - **Cache Memory**: typically attached to the CPU.
 - **Primary storage**: fast, but volatile (RAM).
 - **Secondary storage**: slower, but non-volatile.
 - SSD, HDD or sequential (tape recorders).
- This forms a hierarchy:
 - Secondary transferred to primary and cache and to the CPU at execution time.
- **Databases** are typically stored on secondary storage, but some are stored in main memory.

Figure 17.1

(a) A single-sided disk with read/write hardware.
(b) A disk pack with read/write hardware.



Hardware Overview

- **Buffers** are maintained in primary storage.
 - Temporary stored data from secondary storage.
- **Double buffers** allow:
 - One buffer to written/read to/from the secondary storage (I/O processor).
 - Second buffer to process data at the same time (CPU).

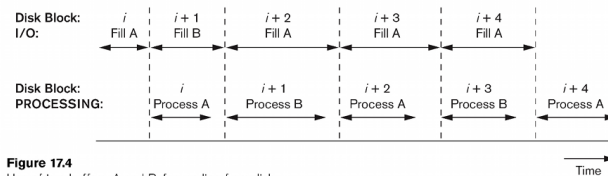


Figure 17.4
Use of two buffers, A and B, for reading from disk.

Placing File Records on Disk

- Stored **records** (tuples) will either have a fixed or variable length.

- **Fixed-length:**

- Records occupy a fixed set of bytes.

- **Variable-Length:**

- Records contain fields that vary in size.

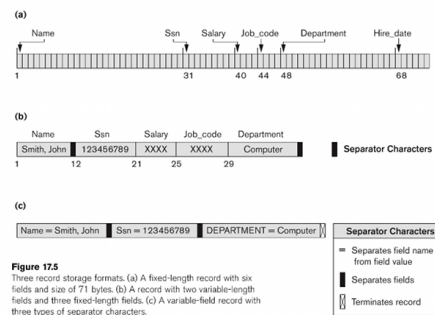
- Data is processed in blocked sized units with R bytes and B block size:

- Fixed length blocking factor:

- $bfr = \lfloor B/R \rfloor$

- Unused space given by:

- $B - (bfr \times R)$



Typical File Operations

- File **I/O functions** are encapsulated within SQL commands with some lower level access.
 - The DBMS will parse the SQL statements.
 - Convert them to an internal representation.
 - Construct an execution plan using the lower-level file operations.
- **Basic operations:** Open, Close, Reset, Read, Find, FindNext.
- **High level:** Insert, Modify, Delete.

Files of Ordered / Unordered Records

- **Records** can be stored:
 - **Unordered:** order of insertion.
 - Insertion is efficient (block data retrieved and re-written)
 - Delete leaves blank space or utilises a *delete-bit*.
 - Linear Search function (fixed length records are useful).
 - Complexity: $O(n)$
 - **Ordered:** based on a key attribute.
 - Insertion is expansive (resorting required).
 - Delete and update are also expansive.
 - Binary search function.
 - Complexity: $O(\log n)$

Algorithm 17.1. Binary Search on an Ordering Key of a Disk File

$l \leftarrow 1$; $u \leftarrow b$; (* b is the number of file blocks *)

while ($u \geq l$) do

begin $i \leftarrow (l + u) \text{ div } 2$;

 read block i of the file into the buffer;

 if $K < (\text{ordering key field value of the first record in block } i)$

 then $u \leftarrow i - 1$

 else if $K > (\text{ordering key field value of the last record in block } i)$

 then $l \leftarrow i + 1$

 else if the record with ordering key field value = K is in the buffer

 then goto found

 else goto notfound;

end;

goto notfound;

Hashing Techniques

- A **Hashing** function provides implicit ordering:

- Maps a **key** attribute to a disk location.
- This attribute is called the **hash key**
- Search algorithm:

- Complexity: $O(1)$

- Example:

- Memory locations 0 to $M-1$.
- Hash key K .
- A hashing function could be:

- $h(K) = K \bmod M$

- **Non-numeric** keys need to be converted to a number first.

Algorithm 17.2. Two simple hashing algorithms: (a) Applying the mod hash function to a character string K . (b) Collision resolution by open addressing.

```
(a) temp ← 1;
   for i ← 1 to 20 do temp ← temp * code(K[i]) mod M;
   hash_address ← temp mod M;

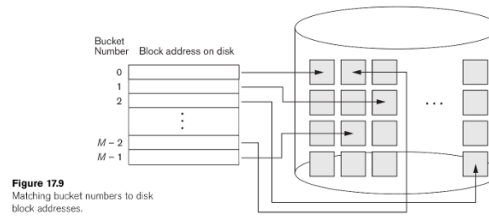
(b) i ← hash_address(K); a ← i;
   if location i is occupied
   then begin i ← (i + 1) mod M;
        while (i ≠ a) and location i is occupied
        do i ← (i + 1) mod M;
        if (i = a) then all positions are full
        else new_hash_address ← i;
   end;
```

Hashing Techniques

- **Collisions:** when different hash keys map to the same location.
- Three solutions for resolving collisions:
 - ***Open Addressing:***
 - Subsequent position checks until an unallocated disk location is found.
 - ***Chaining:***
 - Overflow locations are used to resolve collisions.
 - Typically utilises a *linked-list* at the hash address.
 - ***Multiple hashing:***
 - Program runs a second hash function to find another location.
 - Subsequent collisions are resolved with open addressing.

Hashing Techniques

- Hashing can be:
 - **External hashing:**
 - Disk file level.
 - Address space is made up of **buckets**.
 - Each bucket points to multiple records.
 - Disk blocks can be:
 - Contiguous or single block.
 - File header table maintains order.
 - **Internal hashing:**
 - In program - e.g. arrays.
 - Order preserving.



This is an example of static hashing.

Hashing Techniques

- **Static hashing:**
 - Utilises a fixed number of storage slots (e.g. buckets).
 - Results in wasted storage locations.
 - **Overflow** locations are created if fixed number is exceeded.
 - Requires pointers to additional storage locations.

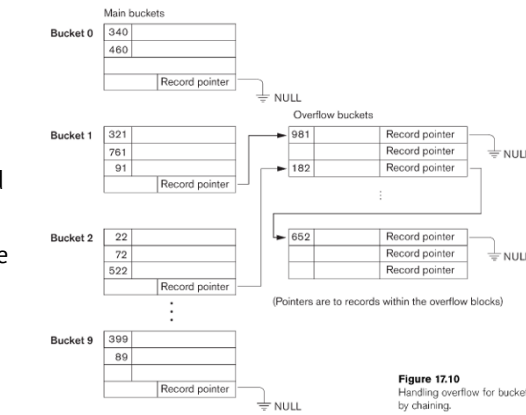


Figure 17.10
Handling overflow for buckets
by chaining.

Hashing Techniques

- Dynamic file expansion allows variable address space.
 - **Extendible** Hashing.
 - **Dynamic** Hashing.
- **Extendible Hashing:**
 - Utilises a 2nd array of bucket addresses.
 - The value d indicates global depth.
 - The value d' indicates local depth.
 - The value d can be increased or decreased.
 - E.g. 00,01 can become 010, 011.

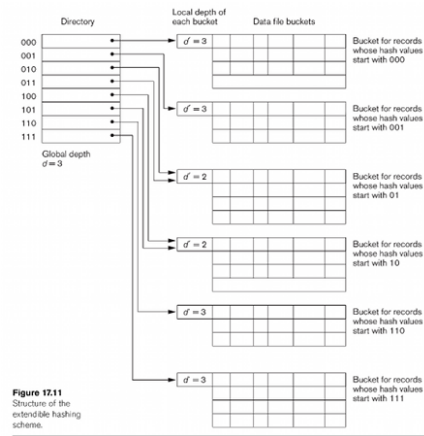
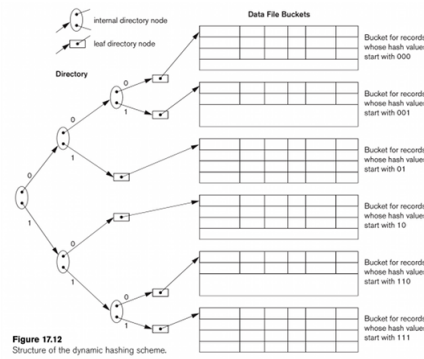


Figure 12.11
Structure of the
extendible hashing
scheme.

Hashing Techniques

- **Dynamic Hashing:**
 - A **tree structure** is used in place of the *flat* directory table.
 - Internal nodes have two pointers corresponding to either 0 or 1.
 - **Leaf nodes** have one pointer that links to the destination bucket.
- The algorithms for insertion and deletion involves simply splitting and combining nodes.



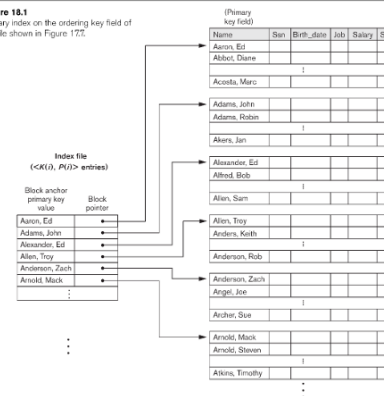
Single Level Indexes

- ***Auxiliary access structures*** called indexes can be used as an alternative or to assist hashing.
- Indexes are designed to speed up the retrieval of individual records.
 - Indexes are a ***Secondary Access Path*** based upon an indexing field(s).
 - Indexes can be **ordered** or **unordered**.
 - E.g. creating an index on employee last name.
- Types of ordered indexes include:
 - **Primary.**
 - **Clustering.**
 - **Secondary.**

Single Level Indexes

- A **primary index**:
 - Ordered file whose records are of fixed length.
 - Consists of two fields:
 - A **key** value - the primary key.
 - A **pointer** to a disk block.
- There is one entry in the index file for each block of data in the data file.
 - Has less entries than the number of records.
 - The first record in each data block is called the *anchor* record.
- Insertion and deletion operations are expensive.

Figure 18.1
Primary index on the ordering key field of
the file shown in Figure 17.2.



Single Level Indexes

- **Indexes** can be characterised as either:
 - **Dense:**
 - A file entry for each of the search **key** values (every record).
 - **Sparse:**
 - A file entry for only a subset of the search values.
 - Refers to multiple records per block.
- The **primary index** is an example of a **sparse** index.
 - It contains an entry for the record at the start of each block.
 - I.e. the **Anchor record**.

Single Level Indexes

- **Clustering indexes:**
 - Another example of a **sparse** index.
 - Ordering is based on a **non-key** field.
 - One entry in the index file for each distinct value.
- The index file's pointer points to the *first* block that contains the corresponding index value.
- Retrieving returns all records that correspond to a distinct key.

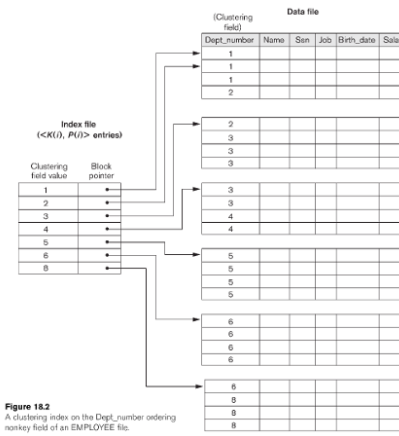
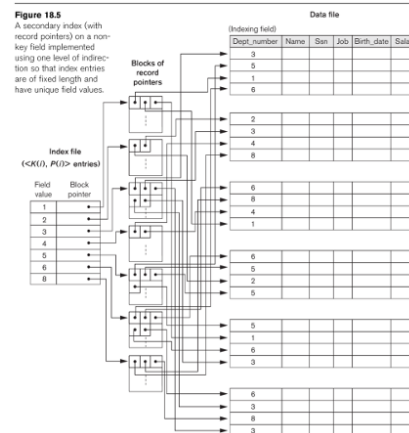


Figure 18.2
A clustering index on the *Dept_number* ordering
nonkey field of an *EMPLOYEE* file.

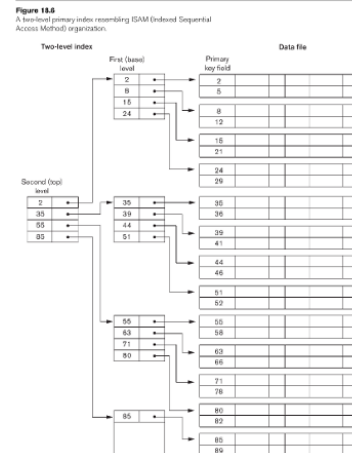
Single Level Indexes

- **Secondary indexes.**
 - Primary access already exists:
 - Created on a candidate key or non-unique attribute.
 - The index file will be ordered by the index field.
- Does not use *Anchors* (non-key).
 - Therefore requires an entry for each record.
 - Implies **dense** indexing.
 - Entry pair will point to the *block* location not the actual record.



Multilevel Indexes

- In single-level indexing there was one entry for each index.
 - Ordered indexing allowed $O(\log_2 b_i)$ block access.
 - Where b_i is the index for each block.
- **Multilevel indexing** files contain a hierarchy of pointers.
 - This reduces the search space by a factor of bfr_i .
 - This process is called **fanning out**.



The Tree Structure

- A **Tree** is a data structure that consists of **nodes**.

- Root node: A.
- Branches: links (pointers) between nodes.
- Leaf nodes: points to a memory location.
 - E, J, C, G, H, K.

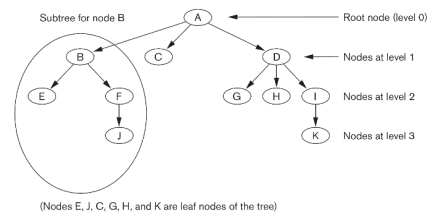


Figure 18.7
A tree data structure that shows an unbalanced tree.

- Multilevel indexing uses a form of **Search Tree**.

Search Trees and B-Trees

- A **Search Tree** provides a guided path to a record.

- **Root node:**

- Points to a subset of records.

- **Leaf node:**

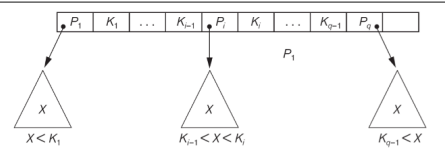
- Points directly to a record.

- Formally: A search tree of **order p** :

- Has $p-1$ search values and p pointers:

- $\langle P_1, K_1, P_2, K_2, P_{q-1}, K_{q-1}, P_q \rangle$
 - where P_i is a pointer to a child node.

Figure 18.8
A node in a search
tree with pointers to
subtrees below it.



Search Trees and B-Trees

- To search for a record we traverse the tree.
 - This involves checking index values at each **child node**.

- Trees can be:

- **Balanced:**

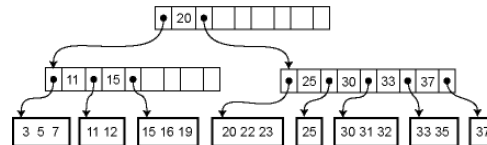
- Leaf nodes have the same number of parent connections.

- **Unbalanced:**

- Connections to leaf nodes are inconsistent.

- **B-Tree (balanced trees):**

- Ensures that search speeds are consistent.
 - This requires re-balancing in the case of deletion and insertion.



Creating Indexes on PostgreSQL Tables

- Indexes can be created on fields within PostgreSQL using the CREATE INDEX clause. With the syntax:

```
CREATE INDEX ON table (index_field) USING method;
```

- CREATE INDEX constructs an index on the specified column(s) of the specified table.
- Indexes are primarily used to enhance database performance.
 - Inappropriate use can result in slower performance.

Creating Indexes on PostgreSQL Tables

- The *index_field* can be specified as a column name.
 - A **Primary key** can be used.
 - **Composite keys** require multicolumn indexing support.
- PostgreSQL uses the **B-tree** structure by default to implement its indexes.
- The *method* option allows different choices:
 - **B-tree**
 - **Hash**
 - **GiST**
 - **GIN**

Summary

- Hardware Overview
- File Records and Operations
- Files of Ordered/Unordered Records
- Hashing Techniques
- Single Level Indexes
- Multilevel Indexes

Reading

- Chapter 16 and 17 from *Fundamentals of Database Systems* by Elmasri and Navathe

