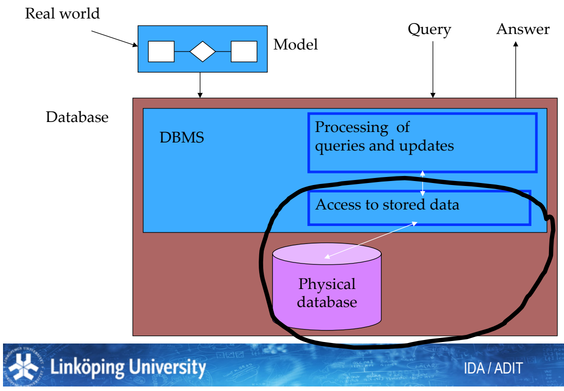
Database Technology
Data structures

Fang Wei-Kleiner

Database system


Real world

Model

Query

Answer

Database

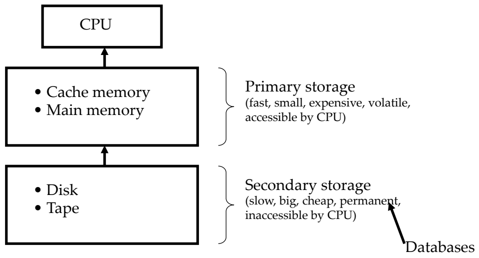
DBMS

Processing of queries and updates

Access to stored data

Physical database

Storage hierarchy


CPU

- Cache memory
- Main memory

Primary storage
(fast, small, expensive, volatile, accessible by CPU)

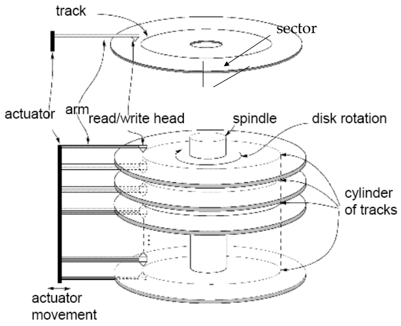
- Disk
- Tape

Secondary storage
(slow, big, cheap, permanent, inaccessible by CPU)

Databases

- Important because it effects query efficiency.

Disk



Disk Storage Devices

- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A **disk pack** contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular **tracks** on each disk **surface**.
  - Track capacities vary typically from 4 to 50 Kbytes or more

---

Disk Storage Devices (cont.)

- A track is divided into smaller **blocks** or **sectors**
- The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed.
  - The block size *B* is fixed for each system.
    - Typical block sizes range from B=512 bytes to B=4096 bytes.
  - Whole blocks are transferred between disk and main memory for processing.

Disk Storage Devices (cont.)

- A **read-write head** moves to the track that contains the block to be transferred.
  - Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:
  - a cylinder number (imaginary collection of tracks of same radius from all recorded surfaces)
  - the track number or surface number (within the cylinder)
  - and block number (within track).
- Reading or writing a disk block is time consuming
  - seek time → 5 – 10 msec
  - rotational delay (depends on revolution per minute) → 3 - 5 msec

---

Disk

- Read/write to disk is a **bottleneck**, i.e.
  - Disk access $\approx 10^{-3}$ sec (9 – 60 milliseconds).
  - Main memory access $\approx 10^{-8}$ sec (50 nanoseconds) .
  - CPU instruction $\approx 10^{-9}$ sec (< 10 nanoseconds)

Files and records

- Data stored in **files**.
- File is a **sequence** of **records (rows)**.
- Record is a set of field values.
- For instance, file = relation, record = entity, and field = attribute.
- Records are allocated to file **blocks**.

Files and records

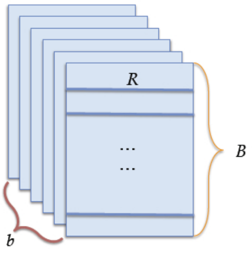
- Let us assume
  - B is the size in bytes of the block.
  - R is the size in bytes of the record.
  - r is the number of records in the file.
- Blocking factor (number of records in each block):

$$bfr = \left\lfloor \frac{B}{R} \right\rfloor$$

- Blocks needed for the file:

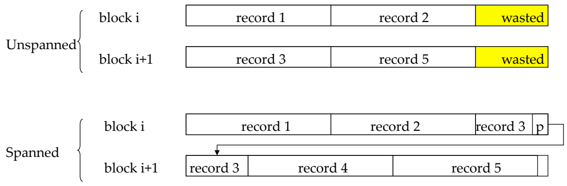
$$b = \left\lceil \frac{r}{bfr} \right\rceil$$

- What is the space wasted per block ?



Files and records

- Wasted space per block = $B - bfr * R$.
- Solution: **Spanned** records.



Unspanned
block i: record 1 | record 2 | wasted
block i+1: record 3 | record 5 | wasted

Spanned
block i: record 1 | record 2 | record 3 | p
block i+1: record 3 | record 4 | record 5

From file blocks to disk blocks

- **Contiguous** allocation: cheap sequential access but expensive record addition. Why ?
- **Linked** allocation: expensive sequential access but cheap record addition. Why ?
- **Linked clusters** allocation.
- **Indexed** allocation.

File organization

- How are the records arranged in the storage?
  - Heap files.
  - Sorted files.
  - Hash files.
- File organization != access method, though related in terms of efficiency.

Heap files

- Records are added to the **end** of the file. Hence,
  - **Cheap** record addition.
  - **Expensive** record retrieval, removal and update, since they imply **linear search**:
    - **Average** case: $\left\lceil \frac{b}{2} \right\rceil$ block accesses.
    - **Worst** case: b block accesses (if it doesn't exist or several exist).
  - Moreover, record removal implies waste of space. So, periodic reorganization.

Sorted files

- Records **ordered** according to some field. So,
  - **Cheap** ordered record retrieval (on the ordering field, otherwise expensive):
    - All the records: access blocks sequentially.
    - Next record: probably in the same block.
    - Random record: binary search, then worst case implies $\lceil \log_2 b \rceil$ block accesses.
  - **Expensive** record addition, but less expensive record deletion (deletion markers + periodic reorganization).
- Is record updating cheap or expensive ?

Internal hash files

- The hash function applies to the hash field and returns the **position** of the record in the **array of records**. E.g.

  position = field **mod** r
- **Collision**: different field values hash to the same position. Solutions:
  - Check subsequent positions until one is empty.
  - Use a second hash function.
  - Put the record in the **overflow** area and link it.

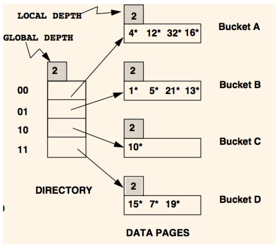
External hash files

- The hash function returns a **bucket** number, where a bucket is one or several contiguous disk blocks. A table converts the bucket number into a disk block address.
- Collisions are typically resolved via overflow area.
- **Cheapest** random record retrieval (search for **equality**).
- **Expensive** ordered record retrieval.
- Is record updating cheap or expensive ?

Extendible hashing

Situation: Bucket (primary bucket) becomes full. Why not re-organize file by *doubling* # of buckets?

- Reading and writing all buckets is expensive!
- *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory*, splitting just the bucket that overflowed!
- Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page*!
- Trick lies in how hash function is adjusted

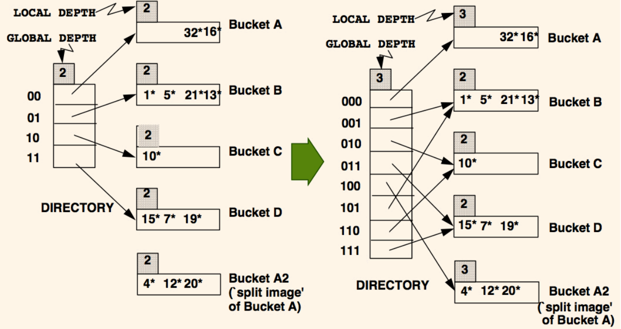


- Directory is array of size 4.
- To find bucket for *r*, take last *global depth* # bits of **h**(*r*); we denote it by **h**(*r*). If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

- **Insert**: If bucket is full, *split* it (*allocate new block, re-distribute*).
- *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

---

*Insert **h(r)=20** (Causes Doubling)*



Extendible hashing (cont.)

- **Extend**: if local depth smaller than global depth then split bucket, else double directory.
- **Shrink**: if local depth smaller than global depth for all the buckets then halve directory.
- **Gain**: no performance degradation due to collisions + space saving.
- At the **cost** of: 2 block accesses per record (directory + data, assuming 1 block per bucket), space for directory, and bucket reorganization.

Linear hashing

- Collisions handled via overflow chain for each bucket.
- Extend when collision
  - Split bucket **n** in two.
  - Distribute blocks in bucket **n** based on K mod **2N**.
  - **n:=n+1**.
- Retrieve: if (K mod N)<n then return K mod N
  else return K mod 2N.
- Shrink based on load factor (=r/(bfr * N)).