Lecture 14: Data Storage

CS 1555: Database Management Systems

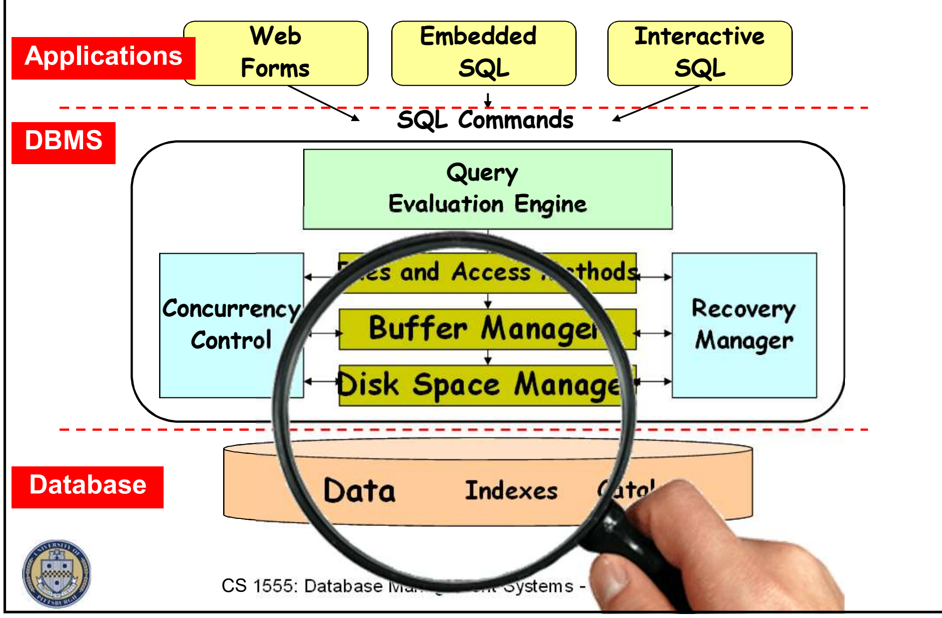
Constantinos Costa

http://db.cs.pitt.edu/courses/cs1555/current.term/

March 21, 2019, 16:00-17:15
University of Pittsburgh, Pittsburgh, PA

Lectures based: P. Chrysanthis & N. Farnan Lectures

Database Management System (DBMS)



Data Storage

- Two DBMS fundamental questions?

1. How do we store and manage very large volumes of data?

2. What representation and data structures best support efficient manipulation of data?
   - RAM model vs. I/O model of Computation

Storage Hierarchy

- Primary Storage: random access; volatile
  - *Cache* - on board or level-2 cache
  - *Main memory*
- Secondary storage: random access; non-volatile
  - Flash-based or Solid State Disk
  - *Magnetic disk*
    - Virtual Memory (Main-memory DBS), File System, DBMS
- Tertiary storage: non-volatile
  - *Optical disk* / juke boxes – random access
  - *Magnetic cartridge* / tape silos – seq. access

Speed; $$

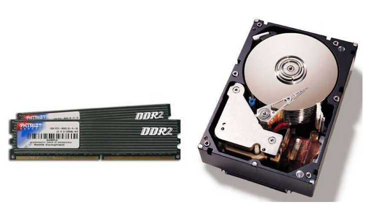
Storage Options

- Disk is slow (order of millisecond),
  but RAM is fast (order of nanosecond)

- Why not store everything in RAM?
  1. Expensive cost
  2. Small capacity
  3. Volatile

Disks and Files

- DBMS stores information on hard disks
- This has major implications on DBMS design:
  - READ: transfer data from disk to RAM
  - WRITE: transfer data from RAM to disk
  - Both are high-cost (I/O) operations (i.e., slow)
    - must be planned carefully!



Magnetic Disks

- Data is stored and retrieved in units called disk **blocks** or **pages**
- Main advantage over tapes:

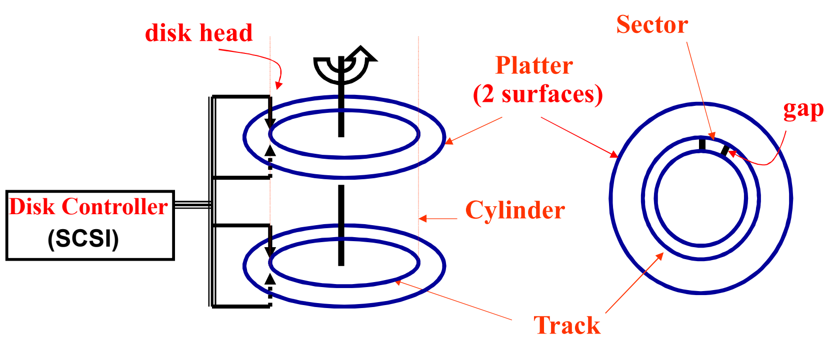
  random access *(vs. sequential)*

- Unlike RAM, time to retrieve a page **varies** depending upon location on disk

Magnetic Disks

- *Disk block* is the unit of transfer: a disk *block* is a continuous sequence of sectors from a single track of one platter.

- Physical address of x: <volume#, platter#, track#, sector#>
- Block address of x: <volume#, sector/block#>



Accessing a Disk Block

- Time to access (read/write) a disk block:

  1. **seek time:**
     - moving arms to position disk head on track
  2. **rotational delay:**
     - waiting for block to rotate under head
  3. **transfer time:**
     - actually moving data to/from disk surface

Performance Measures of Disks

For any block of data,

Average block access time =

seek time + rotational delay + transfer time

- Seek time and rotational delay dominate!

o **Disk Access** = *Seek time* + *Rotation latency*

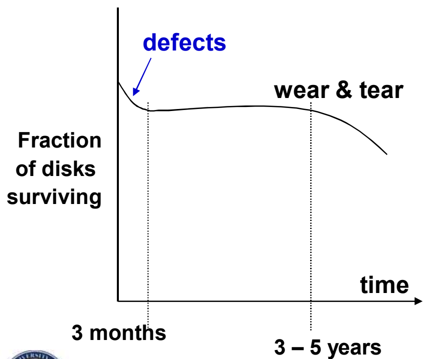
<2000 8 - 20 ms = 2-10 ms + 4-10 ms

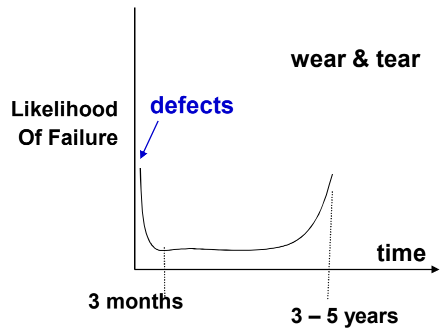
now 2 -10 ms = 1-6 ms + 1-4 ms

o *Data Transfer Rate* = 4-8MB/sec (max 25-40MB/sec)
- Block transfer is less than 0.5 msec
- Multiple disks with share interface can support high rates: 33MB/s(Ultra-DMA), 40MB/s(SCSI-3), 400MB/s(FibreChannel)

Disk Failures

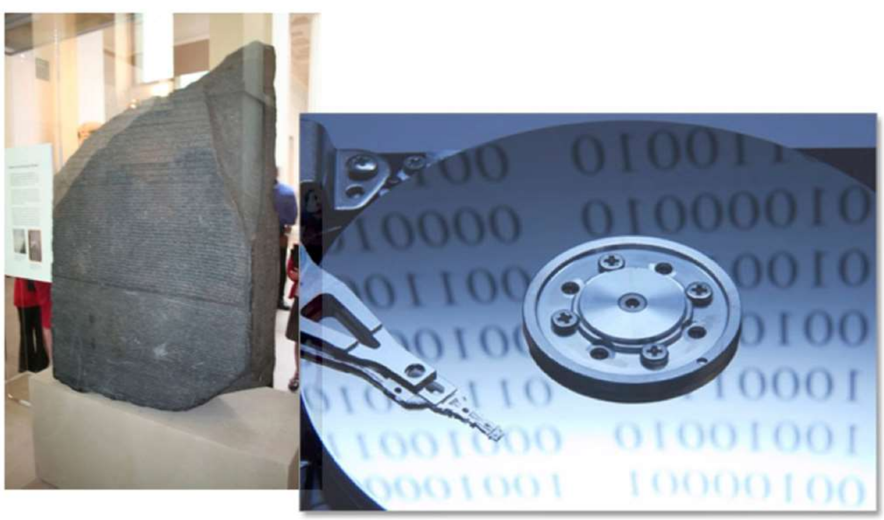
- **Mean Time Between Failure (MTTF)**
  - **Vendor's claim: 30K hrs (3.6 yrs) – 1.2M hrs (136 yrs)**
  - **Practice: 1000 disks with 1.2M hrs MTTF then on average 1 disk will fail in 120 hrs.**





"The BathTub Curve"

But first, how is data stored on disk?



Data Elements

- **Field**: a database attribute (sequence of bytes)
- **Record**: sequence of fields

```
CREATE TABLE Student (
        Sid INTEGER,
        Name CHAR(24),
        Age INTEGER
);
```

0 3	4 27	28 31
Sid	Name	Age

Data Elements

- **Field**: a database attribute (sequence of bytes)
- **Record**: sequence of fields

```
CREATE TABLE Student (
        Sid INTEGER,
        Name CHAR(24),
        Age INTEGER
);
```

0 3	4 27	28 31
Sid	Name	Age

- **Block**: sequence of records
- **File**: sequence of blocks

Blocks & Files

```
CREATE TABLE Student (
        Sid INTEGER,
        Name CHAR(24),
        Age INTEGER
);
```

SID	Name	Age
546007	Peter	18
546100	Bob	19
546107	Ann	21
546207	Jane	20
546240	John	24
546350	Ben	18
546420	Suzy	27
546500	Peter	20

Blocks & Files

```
CREATE TABLE Student (
        Sid INTEGER,
        Name CHAR(24),
        Age INTEGER
);
```

		SID	Name	Age
Block 0	Record 0	546007	Peter	18
	Record 1	546100	Bob	19
	Record 2	546107	Ann	21
	Record 3	546207	Jane	20
Block 1	Record 4	546240	John	24
	Record 5	546350	Ben	18
	Record 6	546420	Suzy	27
	Record 7	546500	Peter	20

Block Header	Record 0	Record 1	Record 2	Record 3

Data on Disk

Address (offset) not stored

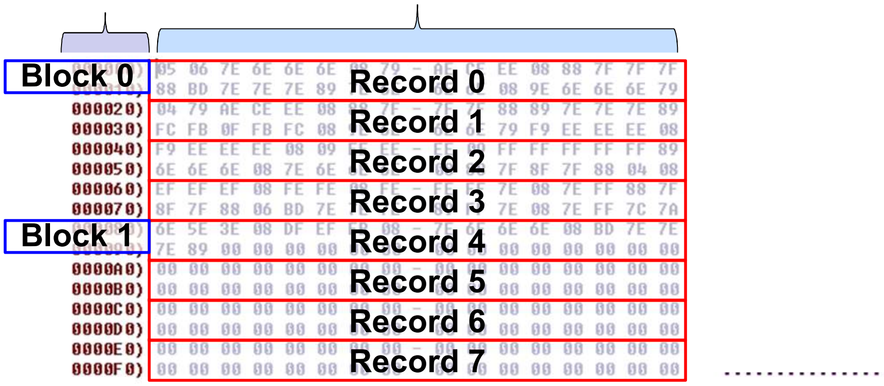
Stored Datas

```
000000) 05 06 7E 6E 6E 6E 08 79 - AE CE EE 08 88 7F 7F 7F
000010) 88 BD 7E 7E 7E 89 7E 6E - 6E 6E 08 9E 6E 6E 6E 79
000020) 04 79 AE CE EE 08 88 7F - 7F 7F 88 89 7E 7E 7E 89
000030) FC FB 0F FB FC 08 9E 6E - 6E 6E 79 F9 EE EE EE 08
000040) F9 EE EE EE 08 09 EE EE - EE 09 FF FF FF FF FF 89
000050) 6E 6E 6E 08 7E 6E 6E 6E - 08 88 7F 8F 7F 88 04 08
000060) EF EF EF 08 FE FE 08 FE - FE FF 7E 08 7E FF 88 7F
000070) 8F 7F 88 06 BD 7E 7E 7E - 89 FF 7E 08 7E FF 7C 7A
000080) 6E 5E 3E 08 DF EF FB 08 - 7E 6E 6E 6E 08 BD 7E 7E
000090) 7E 89 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0000A0) 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0000B0) 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0000C0) 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0000D0) 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0000E0) 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0000F0) 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................
```


Records on Disk

Address (offset) not stored

Stored Data



Improving Access Time ?

- What are the costs in I/O requests?
- I/O cost = *disk access* + queuing delays
- Disk access is reduced with data organization
- Queuing delays are reduced with scheduling

Improving Access Time

- Data access time could be improved by:
  1. Scheduling
  2. Block transfer
  3. Buffering and Prefetching
  4. Cylinder-based Organization
  5. Multiple disks
  6. Record Placement

Queuing Delay

CS 1555: Database Management Systems - Constantinos Costa

Queuing Delay

- **Queuing Delay** occurs when:
  - multiple queries are executed at the same time, and
  - those queries access the same disk at the same time (i.e., send I/O requests)
- The disk head can only serve one request at a time!
- **Solution:**
  - Smart scheduling of serving disk requests
  - Parallel processing (multiple disks organizations)

CS 1555: Database Management Systems - Constantinos Costa

Disk scheduling: Elevator Algorithm

- **Idea** : Schedule reading of requested blocks in the order in which they appear under the head
- **Elevator Algorithm (Scan)** - Works like an elevator
  - The arm moves towards **one direction** serving all requests on the visited tracks
  - Reverses direction when no more pending requests
- **Advantage**:
  - Reduces average access time for unpredictable requests
- **Problem**:
  - It is more effective in many disk-access requests
  - The benefit is not uniform among requests

Variations of Elevator Algorithm

- **Scan or Elevator Algorithm**
  - services both directions and reverses direction when reaches the edge of the disk
- **C-Scan or Circular Elevator Algorithm**
  - all requests are serviced in only one direction
  - more equal performance for all head positions
- **Look**
  - Scan but changes direction when no more pending requests
- **C-Look**
  - Combination of Look and C-Scan
- **FCScan**
  - prevents "arm stickiness" by using two queues (Q1 & Q2)
  - When Scan uses Q1 with old requests, new requests go to Q2

Improving Access Time

- Data access time could be improved by:
  1. Scheduling
  2. Block transfer
  3. Buffering and Prefetching
  4. Cylinder-based Organization
  5. Multiple disks
  6. Record Placement

Block Transfer

- To minimize access time: data is always transferred from/to disks by **blocks** of bytes
- A disk block is an adjacent sequence of sectors from a single track of one platter
- Block size typically ranges from 512 bytes to 4KBytes.

Improving Access Time

- Data access time could be improved by:
  1. Scheduling
  2. Block transfer
  3. Buffering and Prefetching
  4. Cylinder-based Organization
  5. Multiple disks
  6. Record Placement

Buffering / Caching

- **Idea**: Keep as many blocks in main memory as possible to maximize the likelihood that a block of data needed by a transaction is in main memory
- Might run out of memory space ➔ replace blocks!
- Replacement Policies:
  1. LRU (Least Recently Used)
  2. MRU (Most Recently Used)
  3. LFU (Least Frequently Used)
  4. Dual-greedy (e.g., LRU & LFU)
  5. …

Prefetching or Double Buffering

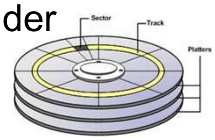
- **Situation**: Needed data are known, but the timing of the request is data-dependent
- **Idea**: speed-up access by pre-loading needed data
- **Cons**:
  - requires extra main memory
  - no help if requests are random (unpredictable)

Improving Access Time

- Data access time could be improved by:
  1. Scheduling
  2. Block transfer
  3. Buffering and Prefetching
  4. Cylinder-based Organization
  5. Multiple disks
  6. Record Placement

Cylinder-based Organization

- ***Observation***: Data such as in relations is likely to be accessed together
- **Idea**: Store such data on the same cylinder
  - Blocks are ***"Next"*** to each other
  - Blocks on a track or on a cylinder effectively involve only the first seek time and first rotational latency



- ***Advantage***:
  - Approaches the theoretical transfer rate
  - Excellent if access can be predicted in advance
  - Only one process/transaction is using the disk

Multiple Disks

- ***Observation***:
  - Disk drives continue to become smaller and cheaper
- **Idea**:
  - Use multiple disks to support **parallel** access
    - Disks with independent heads connected to a single disk controller

  - 2X20 GB drives is faster than a single 40GB drive
    - Bur more expensive…
  - More efficient if they are kept busy!
  - It also enhances system reliability thru redundancy!!

Improving Access Time

- Data access time could be improved by:
  1. Scheduling
  2. Block transfer
  3. Buffering and Prefetching
  4. Cylinder-based Organization
  5. Multiple disks
  6. Record Placement

Multiple Disks: Organizations

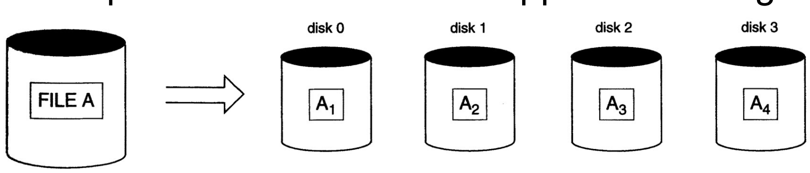
- Data **partitioning** over several disks
  - Pros: increased access rate
  - Problem: if popular data is stored on same disk ➔ request collisions (<u>hot spots</u>)
  - Cons: the cost of several small disks is greater than a single one with the same capacity

- **Mirror** disks: Disks hold identical copies
  - Pros: <u>Doubles</u> read rate (disk 1 **OR** 2)
  - <u>Does not</u> have the problem of request collisions
  - <u>Does not</u> slow down write requests (disk 1 **AND** 2)
  - Cons: Pay cost for two disks to get the storage of one

RAID Technology

RAID Technology

- A major advance in secondary storage technology is represented by the development of **RAID**
  - Redundant Arrays of Inexpensive (independent) Disks
- Acts as a single high-performance large disk
- **Data Striping:** distributes data transparently over multiple disks to make them appear as a single



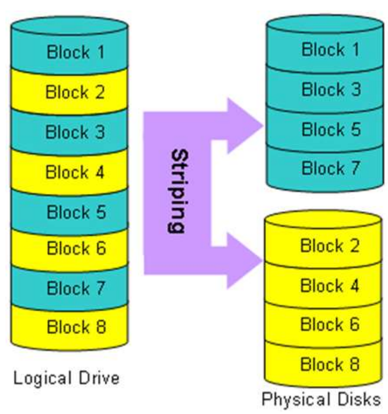
Data Striping

- **Bit-level striping:** Split groups of bits over different disks
  - Example: split each bit of a bytes over 8 disks
  - Bit number *x* is written on disk number *x*
  - Every disk participates in every access
    - every access reads 8 times as many data

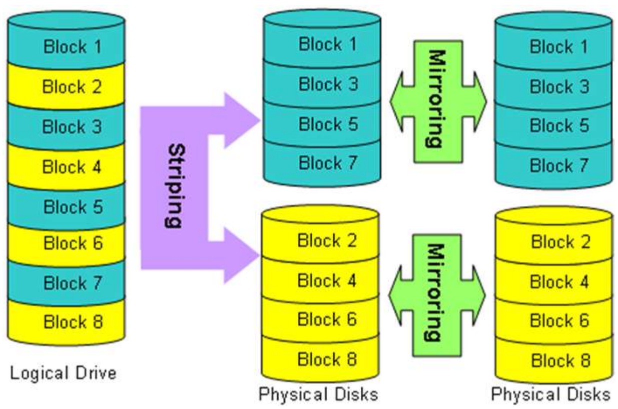
- **Block-level** striping for blocks of a file
- **Sector-level** striping for sectors of a block

- **RAID Goals:**
  - Parallelize accesses so the **access time is reduced**
  - Combining Striping, Mirroring and Reliability ➔ levels of RAID

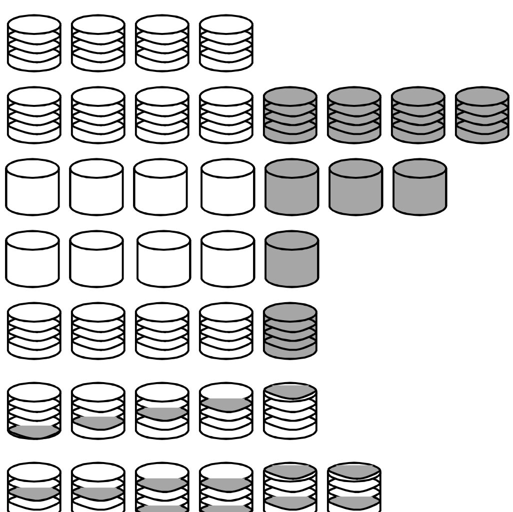
Raid 0



Raid 0-1



Levels of RAID



0: Non-Redundant

1: Mirrored

2: Memory Style ECC

3: Bit-Interleaved Parity

4: Block Interleaved Parity

5: Block-Interleaved Distribution-Parity

6: P+Q Redundancy

Solid State Disk

- **Idea**: A SSD is essentially an array of NAND-flash memory devices which include a controller that emulates a hard disk drive
  - flash translation layer (FTL)
  - single-level cell (SLC) flash

- **Organization (flash):**
  - One or more dies (chips), each 8192 blocks
  - Dies are organized in planes (e.g., 4 planes of 2018 blocks)
  - Blocks is a set of pages 128—256KB (i.e., 32—64 pages)
  - Page is the unit of read and write (program), 512B—4KB
    - 128 bytes are used for metadata
  - Page or Block is the unit of storage system write

Constraints of Solid State Disks

- **Constraints:**
  - **Erase-Constraint** (or erase-before-write): A page cannot be over-written unless the block that contains the page has been erased
  - **Write-Constraint**: Pages must be written out sequentially within a block, from low to high addresses.
    - if p_i is written, p_{i-1} cannot be written until the block is erased
  - **Wear-Constraint**: The number of times a block can be erased is limited – typically 10,000 to 100,000 times.
    - Cleaning and wear-leveling schemes are important
  - **Asymmetry between operations**: Write is more expensive than read (~3 times).

Properties of Solid State Disks

- **Properties**:
  - Random and Sequential Read Access with equal cost
  - Faster than HDD
- parallel requests, interleaving, striping.. 100K – 300K I/Os
  - Energy Efficient.. ~25/750μJ Read/Write, ~220μJ/s Idle
  - Resilient & reliable
    - Shock resistant and extreme temperature fluctuations
    - immune to strong magnetic fields
    - With good wear-leveling can last 10 years
  - Small factor and silent (unless fan is used)

- **Problem**: Higher cost per megabyte of storage
  - Capacity doubles every year

Improving Access Time

- Data access time could be improved by:
  1. Scheduling
  2. Block transfer
  3. Buffering and Prefetching
  4. Cylinder-based Organization
  5. Multiple disks
  6. Record Placement

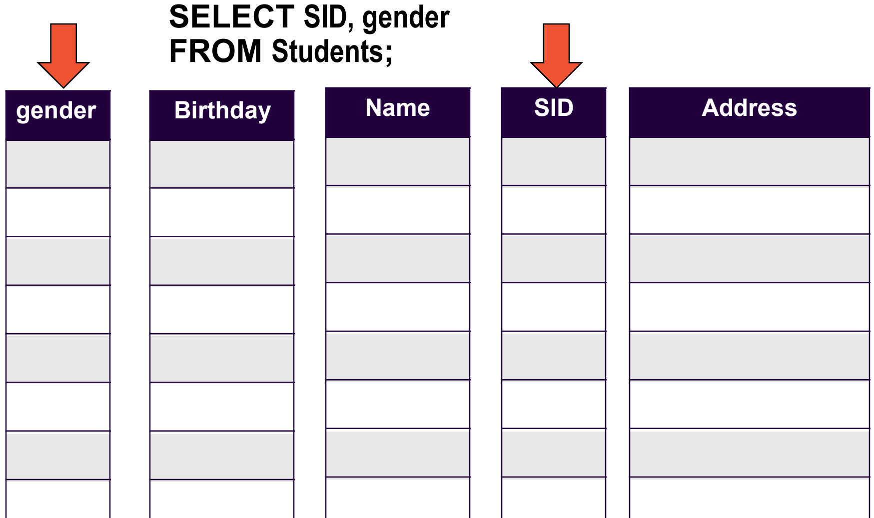
Two Store Approaches

Row Stores

Column Stores

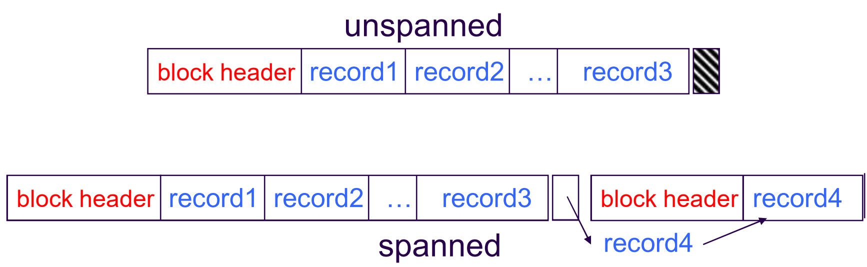
Column Stores

SELECT SID, gender
FROM Students;



Row Stores

- Store fields in one record contiguously on disk
- Use small (e.g., 4K) disk blocks



Record Blocking

- *Block header* includes:
  - Block ID
  - Role info such as data block, index root block, etc.
  - Timestamp
  - Links to other blocks of the same table/file
  - Free-list

- *Blocking factor* ***bfr*** = $\lfloor B/R \rfloor$: records per block
  - B =block size and R= Record size

- For spanned organization,

  Bfr is the average # of records per block

CS 1555: Database Management Systems - Constantinos Costa

Fixed-Length Records

```
CREATE TABLE Student (
  SID         INTEGER
  name        CHAR(16),
  address     CHAR(130),
  gender      CHAR(1),
  birthdate   DATE
)
```

- Fields are stored in sequence as the corresponding attributes are declared
- DATE: 10-char string YYYY-MM-DD
  Fixed-length character string char(10)
  - example: 2002-09-15

Fixed-Length Records - Alignment

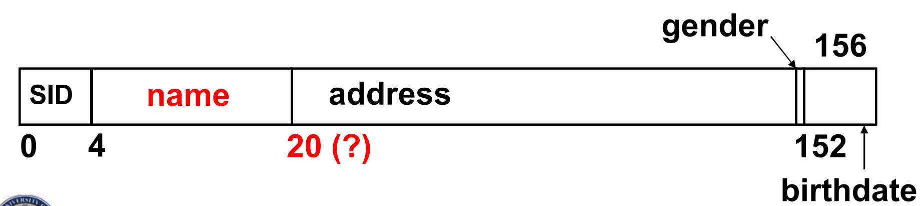
```
CREATE TABLE Student (
  SID         INTEGER,
  name        CHAR(16),
  address     CHAR(120),
  gender      CHAR(1),
  birthdate   DATE
)
```

- Each record within a block starts at a byte that is multiple of 4
- Each field within a record starts at a byte off-set from the beginning of the record that is multiple of 4

Variable-Length Records

```
CREATE TABLE Student (
  SID         INTEGER,
  name        VARCHAR(16),
  address     CHAR(120),
  gender      CHAR(1),
  birthdate   DATE
)
```

- Fields are stored in sequence as the corresponding attributes are declared
- DATE: 10-char string YYYY-MM-DD Fixed-length character string char(10)
  - example: 2002-09-15



Variable-Length Attributes

- example: type **`VARCHAR(16)`**
  - length + data:

10	M	A	B	R	I	N	A	K	I	S

  - special end-of-filed symbol terminated:
    - Any special character that does not appear in any field value, E.g., ¶, § , ■

M	A	B	R	I	N	A	K	I	S	z

  - maximum length:

M	A	B	R	I	N	A	K	I	S						

File Types

- Unordered files
- Ordered files
- Clustered files
- Hash files

File header or descriptor **includes:**
- **field names and their data types**
- **address of the file block on disk**

File header	block1	block2	…	blockN

Unordered Files

- The simplest file structure: records are stored in <u>no</u> particular order
- Also called: **Heap**, Pile, or Random File
- New records are inserted at the **end** of file
  1. The last disk block is copied into buffer (i.e., memory)
  2. New record is added
  3. Block is rewritten back to disk

- Record insertion is quite efficient

Example of Heap File

```
CREATE TABLE deposit (
  account_number  CHAR(10),
  branch_name     CHAR(22),
  balance         REAL
)
```

Block 1	Record 0	A-102	Oakland	400
	Record 1	A-305	Shadyside	350
	Record 2	A-101	Downtown	700
	Record 3	A-222	Squirrel Hill	500
Block 2	Record 4	A-217	Shadyside	900
	Record 5	A-110	Waterfront	340
	Record 6	A-257	Oakland	600
	Record 7	A-403	Downtown	250

Properties of Unordered Files

- File records are inserted at the end of the file or in any file block with free space.
- Thus, insertion is efficient.
- To search for a record, a ***linear search*** through the file records is necessary which is quite expensive.
- Reading the records in order of any field requires sorting the file records.

Ordered Files

- Also called *sequential* files.
- File records are kept sorted by the value of an *ordering* key which has unique value (e.g., primary key)

```
CREATE TABLE deposit (
  account_number   CHAR(10),
  branch_name      CHAR(22),
  balance          REAL
)
```

- Fixed-length records
- 10 + 22 + 8 = 40 bytes

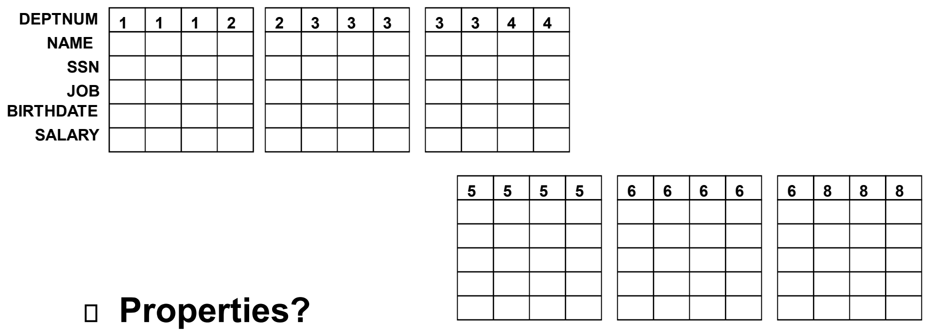
Record 0	A-101	Downtown	700
Record 1	A-102	Oakland	400
Record 2	A-205	Shadyside	350
Record 3	A-217	Shadyside	900
Record 4	A-222	Squirrel Hill	500
Record 5	A-310	Waterfront	340
Record 6	A-357	Oakland	600
Record 7	A-403	Downtown	250

Properties of Ordered Files

- Insertion is expensive: records must be inserted in the correct order.
- Deletion?
- Search for a record on its ordering field value is quite efficient (***binary search algorithm***).
- Search for a record on a non-ordering field?
- Reading the records in order of the ordering field is also quite efficient.
- Reading the records in any order?

Clustered Files

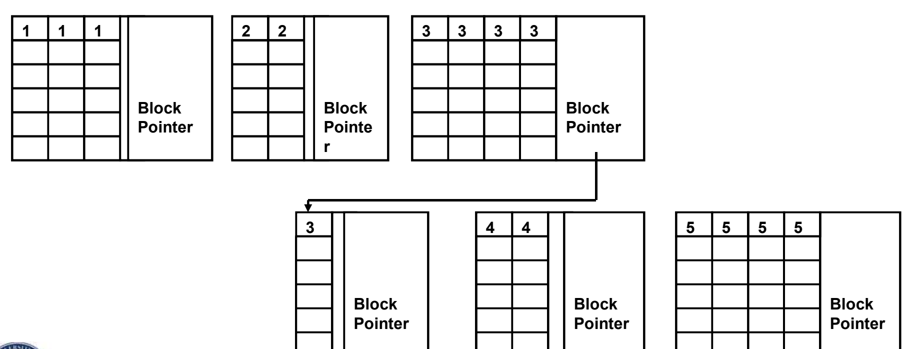
- Order files with ordering field which is **not a key**
- Ordering field has not a unique value



- **Properties?**
- **Purpose ?**

Clustered File: Example 2

- **For efficient insertion, deletion and search, each group of records with different ordering field is stored on separate block**



Hash Files

- Also called **direct** files
- External hashing maps keys to disk blocks
  - Works similar to internal hashing
  - Static hashing
- Dynamic file expansion
  - Linear hashing
  - Extensible hashing

Static Hashing

- Hashing converts the key of a record into an address in which the record is stored.
- An external ***hash*** function maps the key to the relative address of a ***bucket*** in which the record is stored.
  - if a file is allocated ***s*** buckets, the hash function must convert a key ***k*** into the relative address of the block:

$$h(k) \in \{0, ..., s-1\}$$

- A bucket is either one disk block or a cluster of contiguous blocks
- A table stored in the header of the file maps relative bucket numbers to disk block addresses.

Collision

- Insertion of a new record may lead to ***collision***.
  - No space in ***B = h(k)***
- *Probing* or *conflict resolution*:
  - *Open Addressing (Rehashing)*: If bucket h(k) is full, use another hash function until a bucket with a free space is found.

    E.g., *linear probing*

    $$\alpha = \mathbf{h(key)} = \mathbf{key\ mod\ s}$$

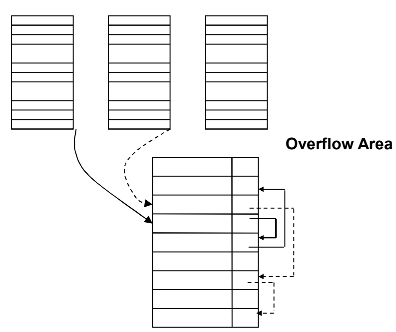
    $$\mathbf{rh(}\alpha\mathbf{)} = \mathbf{h\ (}\alpha\ \mathbf{+1)}$$

    Not a very good technique for databases (Why?)
  - *Chaining*: Use overflow buckets.

Handling of the Overflow Area

- Three approaches:

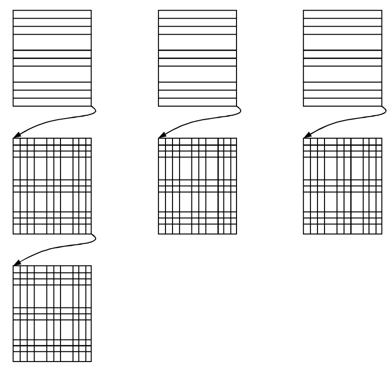
1. ***Common*** overflow area for all blocks in the file.
   - Each block has a pointer to its first record in the overflow area.
   - Records belonging to the same block are linked by pointers.



Handling of the Overflow Area

2. ***Share*** overflow area by some blocks
   - e.g., residing on the same cylinder.

2. ***Individual*** overflow areas
   - Each block has its own overflow
   - Records are not linked



Hashing Functions

- A good hash functions must
  1) be computed efficiently
  2) minimize the number of collisions by spreading keys around the file as evenly and uniform as possible.
- Example of good functions
  - truncation
  - division: h(key) = key mod s
  - Mid-square
  - Folding or partitioning
  - Other ad hoc methods
- Order preserving hash functions:
  - Maintain records in order of hash field values

Pros and Cons of Hashing

- Excellent performance for searching on ***equality on the key*** used for hashing (assuming low density).
- Records are not ordered (heap files).

  ⇒ Any search other than on equality is very expensive (linear search or involves sorting).
- Prediction of total number of buckets is difficult.
  - allocate a large space.
  - estimate a ``reasonable'' size and periodically reorganize.

Dynamic Hashing Methods

- Allow the file size to change as records are added or deleted.
  - Linear Hashing
    - No additional structure
  - Extendible Hashing
  - Binary Hashing

Linear Hashing

- The file size grows linearly, bucket by bucket.
  - The file starts with ***s*** main buckets, where ***s*** is a power of 2, and ***k*** overflow buckets.
  - Buckets are numbered from 0 to ***s-1***
    => initial hashing function $\boldsymbol{h_0(key) = key \bmod s}$.
  - Collisions are handled thru ***chaining***.

- We keep the following info:
  - $\boldsymbol{B_{last}}$: A pointer to the ***current*** last bucket.
    Initially, $\boldsymbol{B_{last} = s-1}$.
  - $\boldsymbol{B_{split}}$: A pointer to the bucket that should be split next.
    Initially, $\boldsymbol{B_{split}} = 0$

Insertion of a Record (Method 1)

- if there is a collision, push tuple to an overflow bucket.
- if there are no overflow buckets available, proceed as follows until one becomes available:
  - A new bucket is appended at the end of the hash table and $\boldsymbol{B_{last} = B_{last} + 1}$
  - Records in $\boldsymbol{B_{split}}$ bucket are hashed again using
    $\boldsymbol{h_1(key) = key}$ mod $\boldsymbol{(2s)}$,
    -- these will either remain in $\boldsymbol{B_{split}}$ or stored in $\boldsymbol{B_{last}}$

  ⇒ *this may free an overflow bucket*.
  - $\boldsymbol{B_{split}}$ becomes $\boldsymbol{B_{split} + 1}$.
- if $\boldsymbol{B_{last}} = \boldsymbol{2s-1}$
  set $\boldsymbol{s = 2s, B_{last} = s-1, B_{split} = 0, h_0(key) = h_1(key)}$
- proceed as above.

Insertion of a Record (Method 2)

- if there is a collision, push tuple to an overflow bucket.
- if there are no overflow buckets available, proceed as follows until one becomes available:
  - A new bucket is appended at the end of the hash table (but B_{last} *does not change as in Method 1)*
  - Records in B_{split} bucket are hashed again using

    $h_1(key) = key \bmod (2s)$,

    -- these will either remain in B_{split} or stored in B_{last}

  ⇒ *this may free an overflow bucket*.

    - B_{split} becomes $B_{split} + 1$.
- if $B_{last} < B_{split}$

  set $s = 2s$, $B_{last} = s-1$, $B_{split} = 0$, $h_0(key)=h_1(key)$
- proceed as above.

Search for a record

- Search for a Record
  - $I = h_0(key)$
  - if $I < B_{split}$ then $I = h_1(key)$
  - search the bucket whose hash value is I (and its overflow, if any).

Variations

- Split a bucket either when there is a need for an overflow bucket or based on some threshold ***u***, e.g., ***u*** > 0.9
- ***u*** is an upper bound of the *file load factor* or number of records that current buckets may store,

  $$l = r/(bfr * N)$$

  r = current number of records, N = current number of buckets/
- Combination can be triggered when, e.g., u < 0.7
- Use a single hashing function:

  A = k mod ***2s***
  if ***A*** > B_{last}
  then ***A = A- s***
  else ***A***

File Types

- Unordered files
- Ordered files
- Clustered files
- Hash files

***File header or descriptor* includes:**

- **field names and their data types**
- **address of the file block on disk**

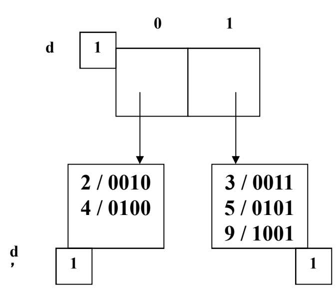
File header	block1	block2	…	blockN

Dynamic Hashing Methods

- Allow the file size to change as records are added or deleted.
  - Linear Hashing
    - No additional structure
  - Extendible Hashing
  - Binary Hashing

Extendible Hashing

- The file is structured into two levels:
  directory and buckets.



Extendible Hashing

- Directory contains only pointers to buckets (no keys).
- Use some hash function to generate a ***pseudokey*** of b-bits (typically b=32)
- Use the first (or last) ***d*** bits to find the offset of the bucket pointer in the directory.
- ***d*** is called the (global) ***directory depth***. It may be stored in the directory.
- In each bucket a local depth ***d'*** is stored indicating the most (or least) significant bits common to all keys in that bucket ($d' \leq d$).

Extendible Hashing – File Growth

- No overflow buckets. New buckets are added one by one.
- When a bucket B with local depth ***d'*** overflows,

  B is split into two buckets and all keys are rehash using ***d'+1*** bits.
- If after a split, ***d' > d***, double the size of the directory (***d=d+1***) to accommodate the growth.