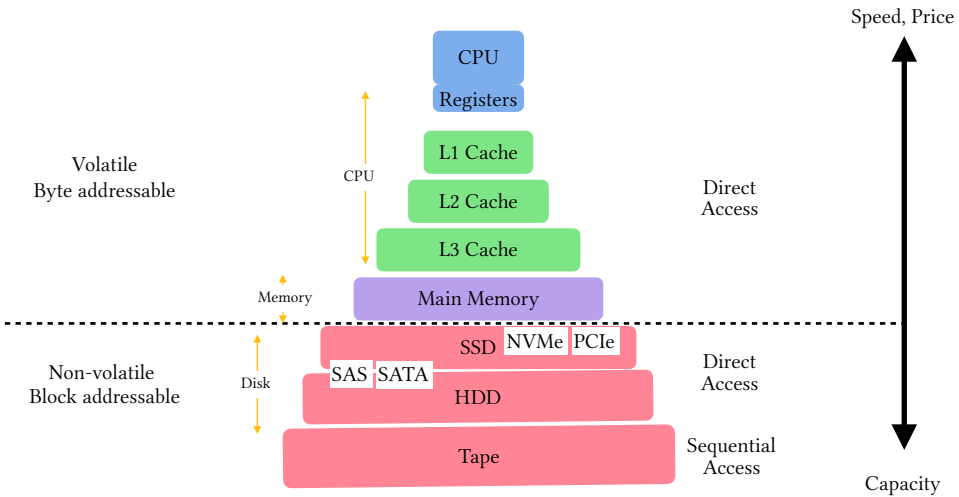
COL362/632 Introduction to Database Management Systems

Database Systems – Storage

Kaustubh Beedkar

Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Memory Hierarchy



Data Storage

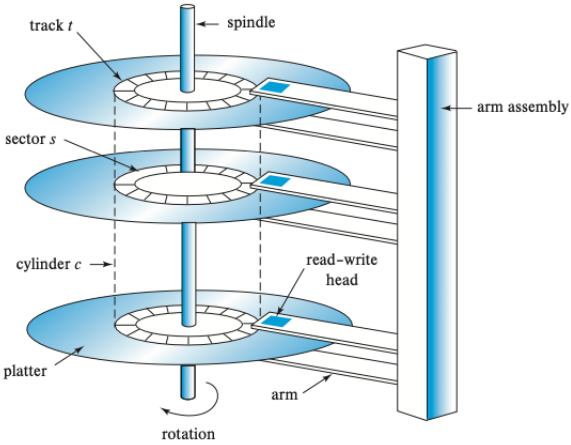
- Storing and accessing data
  - Main memory (fast, but temporary)
  - Disk (slow, but permanent)
- Moving data between disk and memory
  - Buffer manager
- Organizing relational data as files on disk

HDD

- Preferred choice for storing very large volumes of data
- What about SSDs?
  - per byte cost 6–8×!
- **Blocks** Read and write units
  - Where to place data blocks has a huge impact

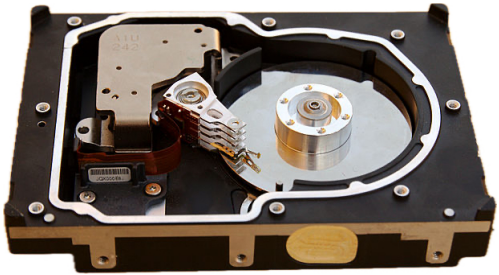
Internals of Disks

- **platter** is the circular hard surface (made of glass or metal) that stores the data
- **spindle** is responsible for rotating the platters (7200 – 15000 Rotations Per Minute (RPM))
- **head** is responsible for reading and writing data
- **arm** is responsible for moving the position of head
- **cylinder** i^{th} track of all platters



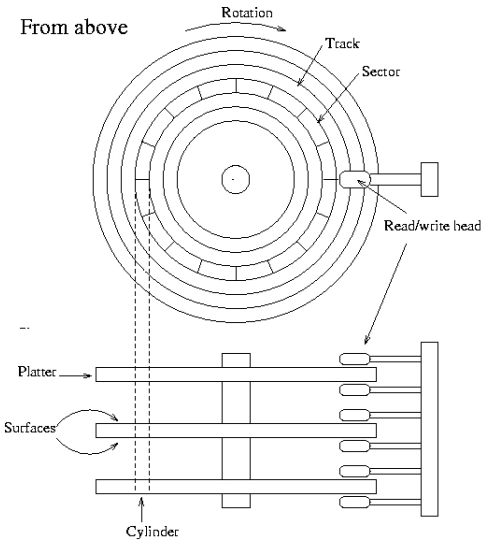
Internals of Disks

- **platter** is the circular hard surface (made of glass or metal) that stores the data
- **spindle** is responsible for rotating the platters (7200 – 15000 Rotations Per Minute (RPM))
- **head** is responsible for reading and writing data
- **arm** is responsible for moving the position of head
- **cylinder** i^{th} track of all platters



Internals of Disks

- disk surface is logically divided in to **tracks**, which are subdived into **sectors**
- **sector** smalled unit of information that can be read or written to disk
- **block** size is a multiple of sector size (fixed)
- At any time only one head can read/write



Reading from- and Writing to Disk

- **Block** logical unit of storage
  - typicall 4KB – 16KB
  - 64KB – 128KB in BigData Systems
  - Data is transferred between disk and memory in **units of blocks**
  - **Page** $\sim$ block in memory
- **access time** = **seek time** + **transfer time** + **rotational latency time**
- **rotational latency time** waiting time for sector to appear under the head
  - typically 0–10ms
  - max: 1 full rotation
  - avg: half rotation

Q : What is the max and average rotational delay for a 7200RPM disk?

Reading from- and Writing to Disk

- **access time** = **seek time** + **transfer time** + **rotational latency time**
- **rotational latency time** waiting time for sector to appear under the head
  - typically 0–10ms
  - max: 1 full rotation
  - avg: half rotation
- **seek time** time taken by the arm to reposition the head
  - typically 2–20ms
- **transfer time** is the time to move blocks to and from platter
  - typically 50 – 200 MB/s

Access Patterns

- **Sequential Access** Successive access requests are for successive block numbers
- **next** block concept
  - same track $\leftarrow$ same cylinder $\leftarrow$ adjacent track/cylinder

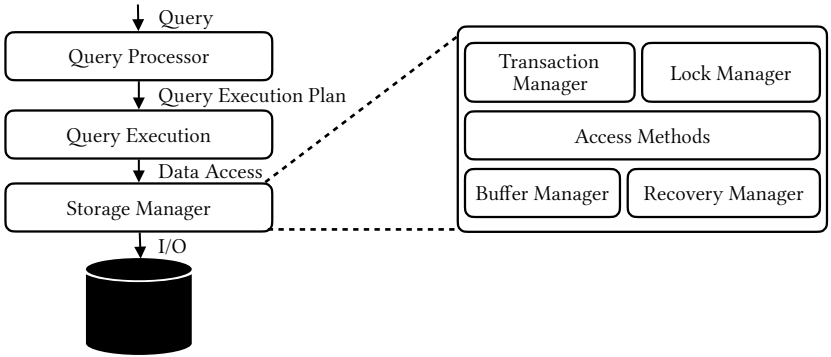
- **Random Access** Successive access requests are for blocks that are randomly located

Blocks in a file should be arranged sequentially on disk to minimize seek and rotational delay

Solid State Drive (SSD)

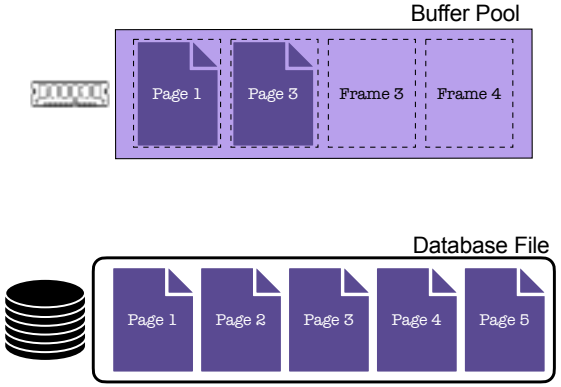
- SSDs are built using flash memory
  - $\sim$ block interface as disks
- no moving parts $\rightarrow$ no seek time and rotational latency time
- higher transfer rates (limited by interconnect)
  - 500MB/s with SATA
  - 3GB/s with NVMe PCIe
- Limited life!

Database Storage



- DBMS stores database in files on disk
- Buffer pool manager takes care of memory management and Disk $\overset{Data}{\longleftrightarrow}$ Memory

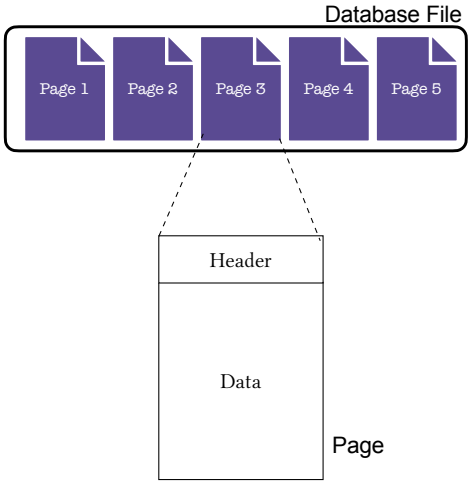
Database Storage



- DBMS stores database in files on disk
- Buffer pool manager takes care of memory management and Disk $\overset{Data}{\longleftrightarrow}$ Memory

File Organization

- Database is stored as a collection of **files**
- Each file is a collection of **pages**
- A page is a fixed-size **block**
  - has an ID
  - can contain tuples, meta-data, indexes, etc.,...



- A table is mapped to a file
- Records (tuples) are mapped to blocks
  - A block contains multiple records
  - A single record could span multiple blocks

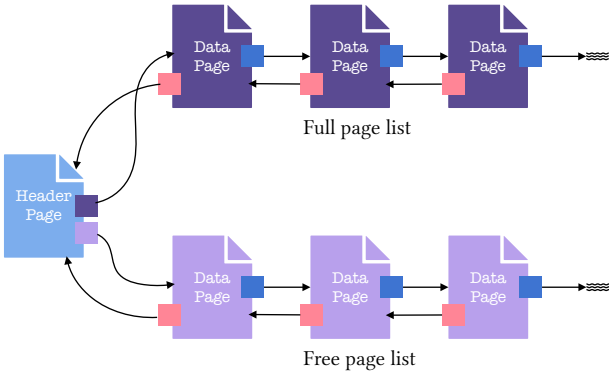
Heap Files

- Records are unordered
- Pages are allocated (deallocated) as file grows (shrinks)

1. Linked List
2. Page Directory

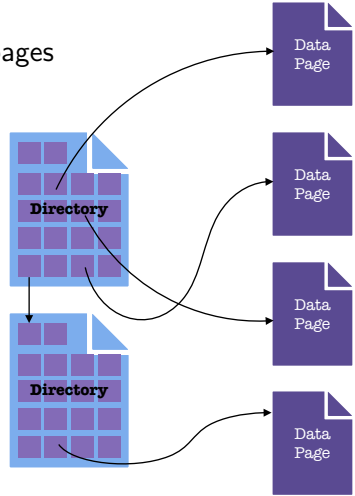
Heap File as Linked List

- **Header page** that stores
  - Heap file name
  - Header page ID
  - Two pointers
    - → Head of free page list
    - → Head of data page list
- Each page has
  - Two pointers `next` and `prev`
  - Data



Heap File as Page Directory

- **Directory** is a special page that keeps tracks of data pages
  - linked list of directory (header) pages
  - location (byte offset) of data pages
  - free space (slots) in each page



Page Organization

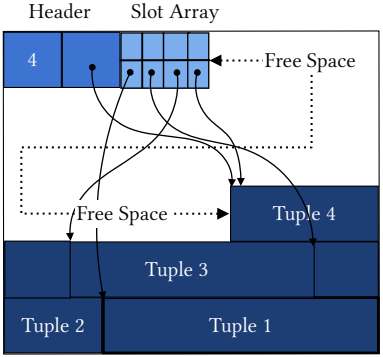
- **Recall** Each file is a collection of records
- File operations are in terms of records or files of records
  - `select * from player where ...`
  - `insert into player values (...)`
  - `update player set first_name = 'Smriti' where player_id = 1`

Page Organization

- Each page contains a **header** containing metadata
  - page size
  - checksum
  - ...
- and **data**
  - Record based
  - Log structured
  - Column stores

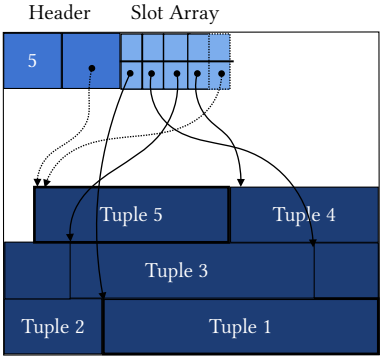
Record-based Organization – Slotted Format

- Page is organized as a collection of **slots**
  - one slot $\sim$ one record
- page header
  - Number of used slots
  - Offset of last slot used
- **slot array**: slot $\mapsto$ record offset
- `record_id` $= \langle page\_ID, slot\# \rangle$
- Records are allocated contigously, starting from the end



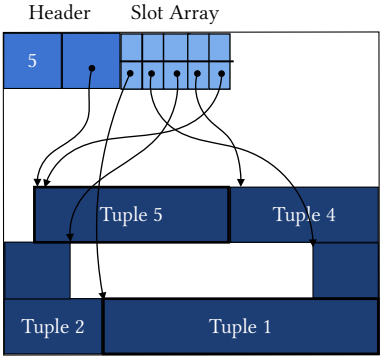
Record-based Organization – Slotted Format

- Page is organized as a collection of **slots**
  - one slot $\sim$ one record
- page header
  - Number of used slots
  - Offset of last slot used
- **slot array**: slot $\mapsto$ record offset
- `record_id` $= \langle page\_ID, slot\# \rangle$
- Records are allocated contigously, starting from the end



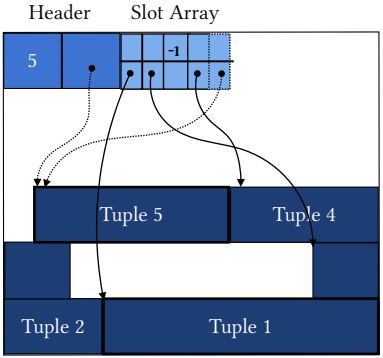
Record-based Organization – Slotted Format

- Page is organized as a collection of **slots**
  - one slot $\sim$ one record
- page header
  - Number of used slots
  - Offset of last slot used
- **slot array**: slot $\mapsto$ record offset
- `record_id` $= \langle page\_ID, slot\# \rangle$
- Records are allocated contigously, starting from the end



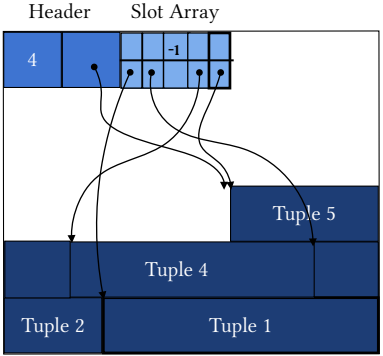
Record-based Organization – Slotted Format

- Page is organized as a collection of **slots**
  - one slot $\sim$ one record
- page header
  - Number of used slots
  - Offset of last slot used
- **slot array**: slot $\mapsto$ record offset
- `record_id` $= \langle page\_ID, slot\# \rangle$
- Records are allocated contigously, starting from the end



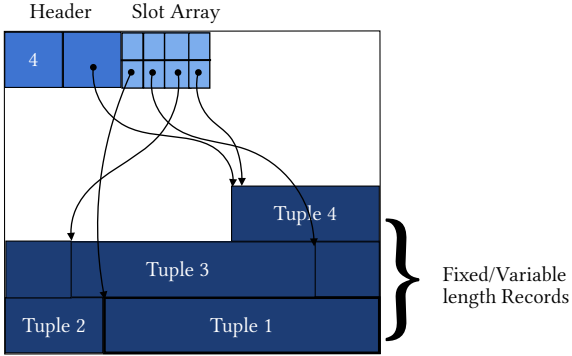
Record-based Organization – Slotted Format

- Page is organized as a collection of **slots**
  - one slot $\sim$ one record
- page header
  - Number of used slots
  - Offset of last slot used
- **slot array**: slot $\mapsto$ record offset
- `record_id` $= \langle page\_ID, slot\# \rangle$
- Records are allocated contigously, starting from the end



Record Layout

- At storage level, a record (tuple) is sequence of bytes
  1. Fixed-length records
  2. Variable-length records

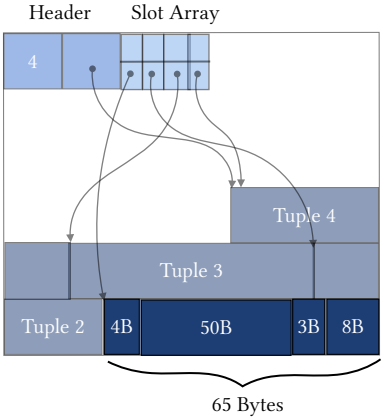


Fixed-length Records

- All records have the same length n
- record i starting from byte $n \times (i - 1)$
- Record may cross blocks
  - do not allow records to cross block boundaries

Example

```
player[
    id: integer,
    name: varchar(50),
    team: varchar(3),
    average: numeric(4,2)
    ]
```



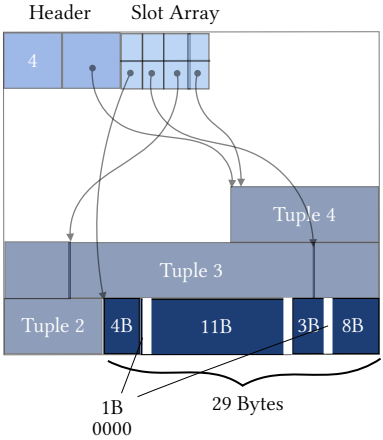
Variable-length Records (1)

- Fields are stored consecutively separated using **delimiters**

Example

```
player[
    id: integer,
    name: varchar(50),
    team: varchar(3),
    average: numeric(4,2)
    ]
```

- (3, 'Virat Kohli', 'RCB', 54.7)

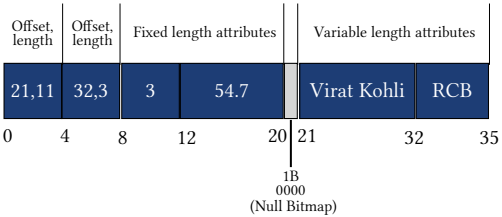


Variable-length Records (2)

- Fields are stored consecutively
- Use fixed size `(offset,length)` pairs in the beginning

Example

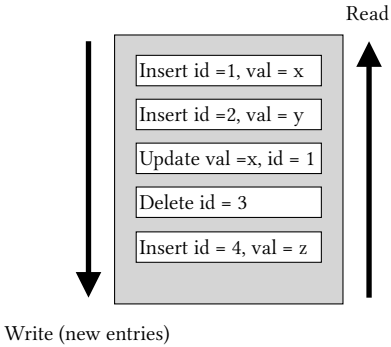
```
player[
    id: integer,
    name: varchar(50),
    team: varchar(3),
    average: numeric(4,2)
    ]
```



- (3, 'Virat Kohli', 'RCB', 54.7)

Log Structured Organization

- Alternative to slotted-array approach
- Store only **log records**
  - i.e, appends log records of low level operations to file
  - Inserts store the entire tuple
  - Deletes mark the tuple as deleted
  - Updates store the delta of modified attributes
- Reading the record
  - Scan backwards to read and recreate the tuple
- Additionally use indexes



Column Stores

- Store data **vertically**
- Each column is mapped to a file
- Ideal of **OLAP** workloads
  - OLAP = Online Analytical Processing
  - OLTP = Online Transaction Processing

Column Stores

- "Reconstructing" a tuple — generally two options
  1. Fixed-length offsets
  2. Embedded record Ids
- Pros
  - Reduced IO
  - Data compression
  - Faster query processing (for certain workloads)
- Cons
  - Slower query processing (e.g., point queries)
  - Slower inserts/deletes/updates