



## COMP 3005 FINAL REVIEW DOC

***PUT QUESTIONS ONLY IN THE QUESTION SECTION BELOW  
AS A TIP TURN ON DOCUMENT OUTLINE IN VIEW MENU***

***TEXTBOOK (CLEARLY BOUGHT AND PURCHASED FOR)***  
***[Database System Concepts 7\(2020, McGraw-Hill Education\).pdf](#)***

RAISE THE BLACK FLAG

Helpful Exam Review Video: **[Database Management Systems Review](#)**

B+-Tree Creation Tool: <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>



# CONTENTS

<b>LINKS</b>	<b>5</b>
Assignment Solutions	5
Midterm Solutions	5
<b>TOPICS</b>	<b>6</b>
<b>Terminologies</b>	<b>6</b>
Relational Algebra	6
Select( $\sigma$ )	6
Project( $\pi$ )	7
Joins	7
<b>SQL</b>	<b>8</b>
DDL	8
Select	9
Insertions	9
Deletions	9
Updates	9
Triggers	9
Functions	9
Views	9
Constraints	9
<b>Entity-Relations</b>	<b>9</b>
<b>Normalization</b>	<b>9</b>
BCNF	9
3NF	9
<b>Storage Organization</b>	<b>9</b>
<b>Physical Storage Media</b>	<b>9</b>
Hierarchy	10
Storage Interfaces	10
Magnetic hard disks	11
DISK CONTROLLER	11
PERFORMANCE MEASURES OF DISKS	12
Flash storage	12
RAID	13
Improvement of Reliability VIA Redundancy	13
Improvement in Performance VIA Parallelism	13
RAID Levels	14
Parity Blocks	15

Choosing Raid Level	15
Optimization of Disk-Block Access	15
Data Storage Architecture	17
File Organization	17
Fixed Length Records	17
Variable Length Records	18
Slotted Page Structure	18
Organization of Records in Files	19
Heap File Organization	20
Free-Space Map	20
Sequential File Organization	21
Partitioning	23
DATA DICTIONARY STORAGE	23
REPRESENTATION OF METADATA	24
STORAGE ACCESS	24
DATABASE BUFFER	24
BUFFER MANAGER	25
PINNED BLOCKS (LITERALLY A SEMAPHORE FROM 3000)	25
SHARED AND EXCLUSIVE LOCKS	25
RULES OF LOCKING	26
ACQUIRING AND RELEASING LOCKS	26
BUFFER REPLACEMENT STRATEGIES	26
COLUMN-ORIENTED STORAGE	27
BENEFITS OF COLUMN-ORIENTED STORAGE	28
DRAWBACKS OF COLUMN-ORIENTED STORAGE	28
HYBRID ROW/COLUMN STORES	29
STORAGE ORGANIZATION IN MAIN-MEMORY DATABASES	29
EXAMPLE	29
Database structures	30
BASIC CONCEPTS	30
INDEX EVALUATION	30
KINDS OF INDICES	30
ORDERED INDICES	31
DENSE INDEX	31
EXAMPLE OF DENSE INDEX (1)	31
EXAMPLE OF DENSE INDEX (2)	31
SPARSE INDEX	32
EXAMPLE OF SPARSE INDEX	32
DENSE VS. SPARSE INDEX	32
GOOD TRADEOFF	33
SECONDARY INDEX	33

CASE FOR MULTILEVEL INDICES	34
MULTILEVEL INDEX(PAGE LEVELS IN 3000)	34
MULTILEVEL INDEX	35
UPDATING THE INDEX (INSERTION)	35
UPDATING THE INDEX (DELETION)	36
INSERTION AND DELETION FOR MULTILEVEL INDICES	37
INDICES ON MULTIPLE KEYS	37
B+ Tree	38
DISADVANTAGES OF INDEX-SEQUENTIAL FILE ORGANIZATION	38
B+-TREE INDEX	38
B+-TREE NODE STRUCTURE	38
LEAF NODES IN B+-TREES	38
NON-LEAF NODES IN B+-TREES	38
B+-TREE STRUCTURE	39
OBSERVATIONS ABOUT B+-TREES	40
QUERIES ON B+-TREES	40
QUERIES ON B+-TREES	41
QUERIES ON B+-TREES (RANGE QUERIES)	41
COST OF QUERYING B+-TREES	41
RANGE QUERIES FOR NON-UNIQUE KEYS	42
UPDATES ON B+-TREES	42
UPDATES ON B+-TREES (INSERTION)	42
SPLITTING A LEAF NODE	42
UPDATES ON B+-TREES (DELETION)	44
DELETION EXAMPLE	45
DELETION EXAMPLE	45
DELETION EXAMPLE	46
Hash Indices	46
COMPLEXITY OF UPDATING B+-TREES	46
B+-TREE FILE ORGANIZATION	47
SECONDARY INDICES AND RECORD RELOCATION	47
SECONDARY INDICES AND RECORD RELOCATION	48
INDEXING STRINGS	48
BULK LOADING IN B+-TREE	48
B-TREE INDEX	49
INDEXING ON FLASH STORAGE	50
INDEXING IN MAIN MEMORY	50
HASH INDICES	51
BUCKET OVERFLOWS	51
USE OF HASH INDICES	51
MULTIPLE-KEY ACCESS	51

INDICES ON MULTIPLE KEYS	52
<b>Questions</b>	<b>53</b>
Relational Algebra	53
SQL	53
Entity-Relations	53
Normalization	53
Storage Organization Raid Questions	54
Database structures	54

## LINKS

### Assignment Solutions

- Assignment 1:  
[https://culearn.carleton.ca/moodle/pluginfile.php/3719883/mod\\_resource/content/1/A1\\_Solution.pdf](https://culearn.carleton.ca/moodle/pluginfile.php/3719883/mod_resource/content/1/A1_Solution.pdf)
- Assignment 2:  
[https://culearn.carleton.ca/moodle/pluginfile.php/3727106/mod\\_resource/content/1/A2\\_Solution.pdf](https://culearn.carleton.ca/moodle/pluginfile.php/3727106/mod_resource/content/1/A2_Solution.pdf)
- Assignment 3:  
[https://culearn.carleton.ca/moodle/pluginfile.php/3763060/mod\\_resource/content/1/A3.pdf](https://culearn.carleton.ca/moodle/pluginfile.php/3763060/mod_resource/content/1/A3.pdf)
- Assignment 4:  
[https://culearn.carleton.ca/moodle/pluginfile.php/3793871/mod\\_resource/content/3/A4\\_Solution.pdf](https://culearn.carleton.ca/moodle/pluginfile.php/3793871/mod_resource/content/3/A4_Solution.pdf)

### Midterm Solutions

- Midterm (Prev Semester):  
<https://drive.google.com/open?id=10SAWMdgB0upfNWzlb6JslqnxWvIUW08t>
- Midterm Winter 2020:  
[https://culearn.carleton.ca/moodle/pluginfile.php/3752399/mod\\_resource/content/1/Midterm.pdf](https://culearn.carleton.ca/moodle/pluginfile.php/3752399/mod_resource/content/1/Midterm.pdf)

# TOPICS

---

## Terminologies

*Note: This will not be tested, but make sure you know what they mean for other sections (i.e. you need to know what a superkey is to normalize a schema).*

- **Relation**  
A table
- **Relation Instance**  
Specific instance of a relation
- **Database Schema**  
Overall design of the database
- **Database Instance**  
Specific instance of a database
- **Tuple**  
Row of a table
- **Attribute**  
Column of a table, identifies a value
- **Superkey**  
One or more attributes that uniquely identify a tuple
- **Candidate key**  
Minimal set of attributes in a superkey that preserves the uniqueness of a tuple, can have multiple candidate keys.
- **Primary key**  
A candidate key chosen to be used as the relation identifier key.
- **Foreign Key**  
An attribute in a relation that is a primary key in another relation

## Relational Algebra

### Select( $\sigma$ )

The SELECT operation is used for selecting a subset of the tuples according to a given selection condition. Sigma( $\sigma$ ) Symbol denotes it. It is used as an expression to choose tuples which meet the selection condition. Select operation selects tuples that satisfy a given predicate.

(Caveman English): you grab something specific from the table :)

Examples:

$\sigma$  sales > 50000 (Customers)

## Project( $\pi$ )

The projection eliminates all attributes of the input relation but those mentioned in the projection list. The projection method defines a relation that contains a vertical subset of Relation. This helps to extract the values of specified attributes to eliminate duplicate values. ( $\pi$ ) The symbol used to choose attributes from a relation.

(Caveman English): This operation helps you to keep specific columns from a relation and discards the other columns.

Examples:

## Set Difference(-)

The result of  $A - B$ , is a relation which includes all tuples that are in A but not in B.

Examples:

## Cartesian Product(X)

Cartesian operation is helpful to merge columns from two relations

Examples:

## Joins

### Inner Join

Inner join, includes only those tuples that satisfy the matching criteria.

Examples:

### Natural Join( $\bowtie$ )

Natural join can only be performed if there is a common attribute (column) between the relations.

Examples:

### OUTER JOIN

In an outer join, along with tuples that satisfy the matching criteria, we also include some or all tuples that do not match the criteria.

Examples:

### Left Outer Join

In the left outer join, operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.

Examples:

### Right Outer Join

In the right outer join, operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.

Examples:



## Full Outer Join

In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.

Examples:

### Employee

Name (PKEY)	company	salary
Mike Myers	Apple	52000
Ahmed El Roby	Carleton	80000
Thomas Edrich	Apple	120000
Joe Biden	Walmart	32000
Suzane Riders	Carleton	72000

### Company

Name (PKEY)	Location
Apple	California
Carleton	Ottawa
Walmart	Florida

## SQL

### DDL

#### SQL Query

##### **Data Definition Language:**

Use to define schemas, deleting, modifying relations. Essentially, the blueprint.

##### **Create a relation:**

```
Create table employee
(name          varchar(20),
Company       varchar(20),
Salary        numeric(8,2),
Primary key (name),
Foreign key (company) references company (name)
);
```

##### **Delete all tuples:**

```
Delete from employee;
```

**Removing a relation:**

Drop table employee;

Keeping relation but removing all tuples:

Delete from course:

Adding a new attribute to a relation

Alter table course add capacity numeric (3.0)

```
(select course_id
from section
where semester = 'Fall' and year = 2017)
union
(select course_id
from section
where semester = 'Spring' and year = 2018);
```

**Basic aggregation**

```
Select avg (salary)
From instructor
Where dept_name = 'Comp. Sci'
```

**Group by aggregation**

```
Select dept_name, avg(salary) as avg_salary
From instructor
Group by dept_name;
```

```
//incorrect below
Select dept_name, ID, avg(salary)
From instructor
Group by dept_name
```

**Group by with having**

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name
having avg(salary) > 42000;
```

dept_name
Biology
Comp Sci

Set membership using in and not in

```
select distinct course_id
from section
where semester = 'Fall' and year = 2017 and
       course_id in (select course_id
                     from section
                     where semester = 'Spring' and year = 2018);
```

With clause

- **with** provides a way of defining a temporary relation that is not actually stored in the database
- Find the departments with the maximum budget

```
with max_budget (value) as
    (select max(budget)
     from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

## Deletions

- Delete instructors whose departments are in the Watson building

```
delete from instructor
where dept_name in (select dept_name
                    from department
                    where building = 'Watson');
```

- Delete instructors with salary below the average at the university

```
delete from instructor
where salary < (select avg (salary)
               from instructor);
```

## Insertions

```
insert into course
values('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

```
insert into instructor
select ID, name, dept_name, 18000
from student
where dept_name = 'Music' and tot_cred > 144;
```

## Updates

- Change some values in a tuple without changing all values in the tuple
- Raise time! Increase the salaries by 5%

```
update instructor  
set salary = salary * 1.05;
```

- Increase salary for instructors with a salary of less than \$70,000

```
update instructor  
set salary = salary * 1.05  
where salary < 70000;
```

## Triggers

## Functions

## Views

## Constraints

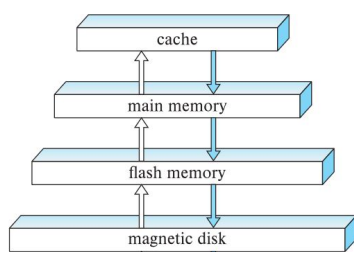
## Entity-Relations

## Normalization

## BCNF

## 3NF

## Storage Organization



## Physical Storage Media

### Cache

- Fastest
- Most costly
- Shouldn't be concerned with managing cache
- Database implementors do pay attention to cache effects

### Main Memory

- Too small, or too expensive to store the entire database for very large databases
- Volatile
- Tens of gigabytes to thousands of gigabytes

### Flash Memory

- A Solid-state drive (SSD) uses flash memory internally
- Can be used to replace magnetic disks
- Non-volatile
- Terabytes

### Magnetic Disk

- Primary medium for long-term storage
- Also referred to as hard disk drive (HDD)
- Non-volatile
- To access data, data must be moved to main memory
- Terabytes

### Optical Disk

- Digital video disk (DVD)
- Blu-ray DVD can reach 128 gigabytes

### Magnetic Tapes

- Primarily for archival data (stored safely for a long period of time)
- Access is much slower
- Must be accessed sequentially

## Hierarchy

Primary Storage: Fastest media, but volatile (cache, main memory)

Secondary Storage: moderately fast access time (flash memory)

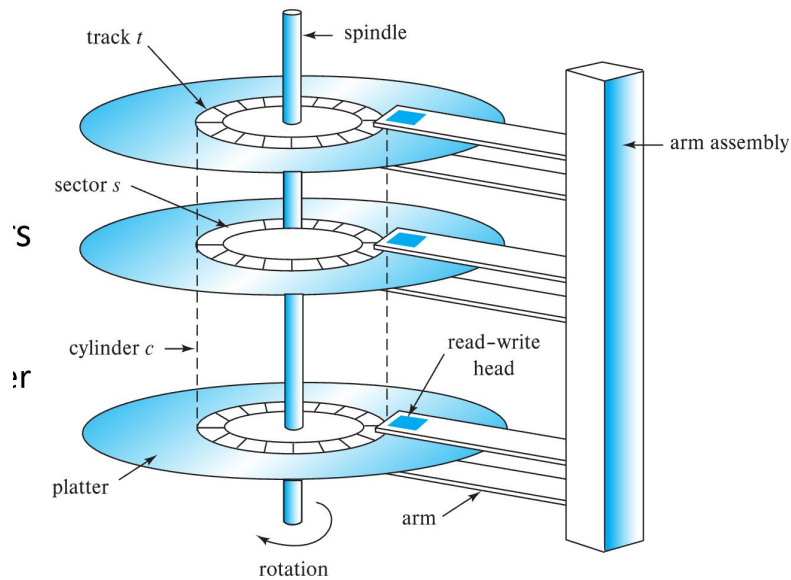
Tertiary Storage: Non-volatile, and slow access time (magnetic tape, optical storage)

## Storage Interfaces

- Disk interface standards families
  - SATA (Serial Advanced Technology Attachment)
    - SATA-3 supports data transfer speeds of up to 6 gigabits/second
    - Actual cable
  - SAS (Serial Attached SCSI (Small Computer System Interface))
    - SAS version 3 supports 12 gigabits/second

- typically used on servers
  - NVMe (Non-Volatile Memory Express) interface
    - built to better support SSDs
    - Supports data transfer rates of up to 24 gigabits/sec
    - Connects directly to motherboard
- Disks are usually connected directly to the computer system
- Can also be situated remotely and connect via high-speed network

## Magnetic hard disks



- Surface of platter divided into circular tracks
  - Over 50K-100K tracks per platter
- Each track is divided into sectors
- The smallest unit of data
  - Typically 512 bytes
  - Inner tracks: 500-1000 sectors per track
  - Outer tracks: 1000-2000 sectors per track
- To read/write a sector:
  - Disk arm swings to position on right track
  - Platter spins continually and data is read/written as sectors pass under head
- Multiple platters on a single spindle
- One head per platter, mounted on the arm assembly
- Heads move together
  - When head on the  $i$ th track on one platter, all other heads are also on the  $i$ th track of their platters
  - These tracks are called the  $i$ th cylinder

## DISK CONTROLLER

- Interfaces between the computer system and actual hardware of the disk drive
- Accepts high-level commands to read or write a sector
- Initiates actions such as moving the disk arm to the right track and start reading or writing
- Computes and attaches checksums to each sector to verify data is read correctly
  - If wrong checksum is computed, controller will retry the read. If the error continues, the controller will signal a read failure
- Remapping of bad sectors
  - If the controller detects that a sector is damaged, it can logically map the sector to a different physical location (from a pool of extra sectors set aside for this purpose)
  - The remapping is noted on disk so that the write is carried out on the new location

## PERFORMANCE MEASURES OF DISKS

- Access time: The time it takes from when a read or write request is issued to when the data transfer begins
  - Seek time: The time it takes to reposition the arm over the correct track
    - 4 to 10 milliseconds on typical disks
  - Rotational latency: The time it takes for the sector to be accessed to appear under the head
    - 4 to 11 milliseconds on typical disks
- Access time = Seek time + Rotational latency
- Data-transfer rate: The rate at which data can be retrieved from or stored to disk
  - 25 to 200 MB per second max rate
  - Transfer rate for inner tracks is lower than the max rate (because they have fewer sectors)
- Disk block (page): Logical unit for storage allocation and retrieval
  - 4 to 16 KB
  - Smaller blocks More transfers from disk
  - Larger blocks More space wasted due to partially filled blocks
- Sequential access pattern
  - Successive requests are for successive disk blocks
  - Disk seek required only for first block
- Random access pattern
  - Successive requests are for blocks that can be anywhere on disk
  - Each access requires a seek
  - Transfer rates are low since a lot of time is wasted in seeks
- I/O operations per second (IOPS)
  - Number of random block reads that a disk can support per second
  - 50 to 200 IOPS on current generation magnetic disks
- Mean time to failure (MTTF): The average time the disk is expected to run continuously without any failure
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low
  - MTTF decreases as disk ages



## Flash storage

- NAND flash
  - Used widely for storage
  - Reads a page at a time (512 - 4096 bytes)
  - Page can be written only once, must delete the page to rewrite
- Solid State Disks
  - Data divided into blocks, but store data on multiple flash devices internally.
  - SATA
    - 500 mb/s
  - NVMe PCI
    - 3 gb/s
- Erasing:
  - Units of “erase block”
  - Takes 2 - 5 milliseconds
  - Erase block size: 256 kb - 1 mb (64-256 4k pages)
  - To limit slow erase speed, an updated logical page can be remapped to an already erased physical page.

## RAID

- Lord linux links:
  - RAID 0, 1 10: <https://www.youtube.com/watch?v=eE7Bfw9lFfs>
  - RAID 5, 6: <https://www.youtube.com/watch?v=1P8ZecG9iOI>
- Redundant Arrays of Independent Disks
- Disk organization technique to manage a larger number of disks, providing a view of a single disk
- High capacity: Many disks providing larger storage spaces
- High speed: Allows using multiple disks in parallel
- High reliability: Storing data redundantly so data can be recovered in case of failures
- The chance that some disk out of N disks will fail is much higher than the chance that a specific single disk will fail
  - Techniques for using redundancy to avoid data loss are extremely critical in this setting

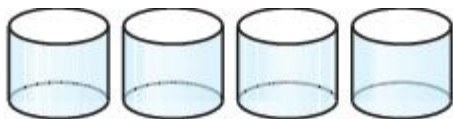
## Improvement of Reliability VIA Redundancy

- Mirroring (Shadowing)
  - Duplicate every disk
  - Any logical disk consists of (at least) two physical disks
  - Every write is carried out on both disks
  - Reads can use either disks
  - If one disk fails, data is still available on the other
  - Data loss will occur only if both the disk and its mirror fail, or if the mirror fails before the first is repaired/replaced
  - Probability of combined event is very small
    - Except for cases of power failure or natural disaster
- Mean time to data loss depends on mean time to failure and mean time to repair

## Improvement in Performance VIA Parallelism

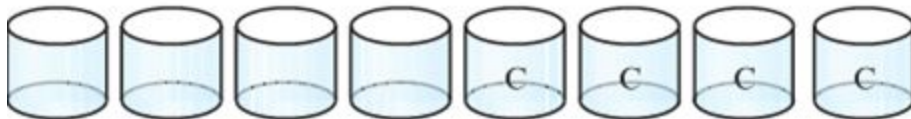
- With disk mirroring, assuming two disks, the rate of handling read requests is doubled, but the transfer rate of each disk is still the same
- Improve transfer rate by striping data across multiple disks
- Bit-level striping: Split the bits of each byte across multiple disks
  - In an array of eight disks, write bit  $i$  of each byte to disk  $i$
  - Each access can read data at eight times the rate of a single disk
  - But average seek/access time is same or worse than for a single disk
- Block-level striping: With  $n$  disks, block  $i$  of a file goes to disk  $(i \bmod n) + 1$ 
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel

## RAID Levels

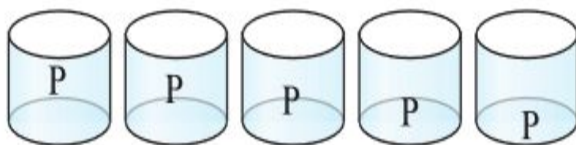


- All following RAID examples show 4 disks worth of data
- RAID Level 0: Disk arrays with striping at the level of blocks, but no redundancies
  - Used in high-performance applications where data loss is not critical

- Raid Level 1: Disk mirroring with block striping (c denotes copy)



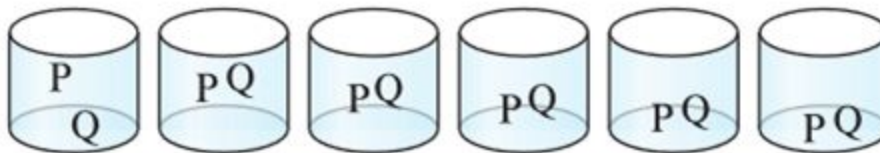
- Raid Level 5 (Block-interleaved distributed parity): Partitions data and parity among all  $N + 1$  disks, rather than storing data in  $N$  disks and parity in 1 disk
  - Block writes occur in parallel if the blocks and their parity blocks are on different disks



- Example:

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

- Raid Level 6 (P + Q Redundancy Scheme): Similar to level 5, but stores two error correction blocks (P, Q) instead of single parity block to guard against multiple disk failures
  - Better reliability than level 5 at a higher cost
  - write is slower than level 6



## Parity Blocks

Parity block : The designation of one or more bits in a block as parity bits used to force the block into a selected parity, either odd or even. (adds 0 or 1 to end )Note: Block parity is used to assist in error detection or correction.

- Some indian guy I trust: <https://www.youtube.com/watch?v=vNPBfq-gKHE>
- Some other dude I trust: <https://www.youtube.com/watch?v=UuUgfCvt9-Q> \*\*\*\*\*
- RAID 5: In N+1 disks, only N are data. The other disk is used to store parity.
- For a set of blocks, a parity block's ith bit is computed as the XOR of the ith bits of all the blocks in the set
- To recover data for a block, compute XOR of the other blocks and the parity block
- When writing data to a block j, the parity block j needs to be updated and written to disk
  - Recompute the parity value using the new values of blocks corresponding to the parity block
  - Using old parity block, old value of current block, and the new value of the current block

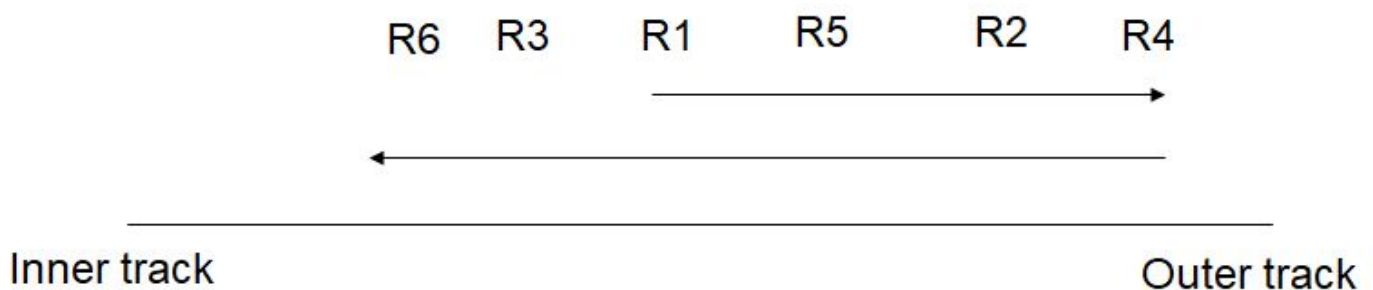
## Choosing Raid Level

- Multiple factors
  - Monetary cost
  - Performance: Number I/O operations per second, and bandwidth during normal operation
  - Performance during failure
  - Performance during rebuild of failed disks
- Raid 0
  - When data safety is not important
- Raid 1

- Better write performance than level 5
- Level 5 requires at least 2 block reads and 2 block writes to write a single block vs. Only 2 block writes for RAID 1
- Raid 5
  - For applications where writes are sequential and large (many blocks), and need large amounts of data storage
- Raid 6
  - Better data protection than RAID 5 since it can tolerate two disk failures
  - slower write than raid 5 but similar read

### Optimization of Disk-Block Access

- Requests for disk I/O are generated by the database system (specifically, query processing component)
- Each request specifies a disk identifier and a logical block number on disk
- Sequential access: Successive requests are for successive block numbers, which are on the same track or on adjacent tracks
- Random Access: Successive requests are for blocks that are randomly located on disk. Each request would require a seek, resulting in a longer access time and lower number of random I/O operations per second
- Buffering
  - Blocks that are read from disk are stored temporarily in an in-memory buffer
  - Done by both OS and DBMS
  - Read-ahead from a track in anticipation that they will be requested soon
  - Good for sequential access, but useless for random access
- Disk-arm-scheduling
  - Re-order block requests so that disk arm movement is minimized
  - Read extra blocks (in buffer)
  - Elevator algorithm



- File organization
  - Organize blocks on disk in a way that corresponds closely to the way we expect data to be accessed
  - If we expect a file to be accessed sequentially, we should ideally keep all the blocks of the file sequentially on adjacent cylinders
  - Large file? OS allocates some number of consecutive blocks (extent) at a time to a file. Different extents may not be adjacent to each other
    - Need one seek per extent

- Fragmentation: Over time, a sequential file that had multiple small appends may become fragmented
  - Make backup of data, then restore disk and try to write back the blocks of each file contiguously
- Contents of main memory are lost in power failure
- Information about database updates must be recorded on disk
- Performance of update-intensive database applications heavily depends on the latency of disk writes
- Use non-volatile random-access memory (NVRAM) to speed up disk writes (Non-volatile write buffers)
- Flash memory is currently the primary medium for non-volatile write buffering
- When the DBMS or OS requests that a block be written to disk, the disk controller writes it to non-volatile write buffer and notifies of a write success
- Write data to their destination on disk later in a way that minimizes disk arm movement

## Data Storage Architecture

- Persistent data is stored on non-volatile storage media (magnetic disk or SSD)
- Both are block structured devices
- Database deal with records
  - Records are usually much smaller than a block
- Most dbs use OS files as an intermediate layer for storing records
- However, to ensure efficient access and better recovery from failures, databases must continue to be aware of blocks

## File Organization

- The database is stored as a collection of files
- Each file has a sequence of records (tuples)
- A record is a sequence of fields (values of attributes)
- One approach: Assume record size is fixed
- Assumption: Records are smaller than a disk block (no record is larger than a block)
  - Realistic for most data-processing applications
  - Larger objects (e.g., images) will be discussed later

## Fixed Length Records

- Store record  $i$  starting from byte  $n * (i-1)$ , where  $n$  is the size of each record
- Example:
  - $i=5$
  - $N = 10$
  - location =  $10 * (5-1) = 10 * 4 = 40$
- Deleting a record:

```
create table instructor
(ID          varchar(5),
name        varchar(20),
dept_name   varchar(20),
salary     numeric(8,2),
primary key (ID),
foreign key (dept_name) references department
);
```

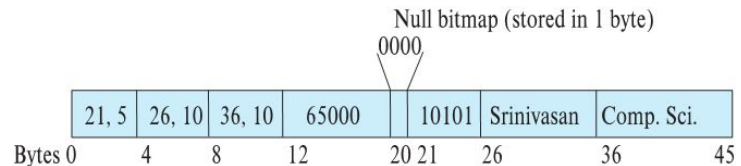
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

- Each record is 53 bytes long (5+20+20+8)
- What not to do:
  - Delete record  $i$ , then move records  $i+1 \dots n$  to  $i \dots n-1$ :  $O(n-i)$ , inefficient to move large number of records
  - Don't move record  $n$  to  $i$ , requires additional block access.
- What to do:
  - Allocate some bytes for a file header that holds the address of the first record whose content was deleted.
  - Each subsequent record holds the address of the next deleted record and so on.

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

## Variable Length Records

- Each record holds two parts:
  - Fixed length information
    - Holds attributes (numeric values, dates, etc.)
    - Information on variable length attributes (varchar)
      - Offset: starting byte
      - Length: how many bytes
  - Contents of variable length attributes



## Slotted Page Structure

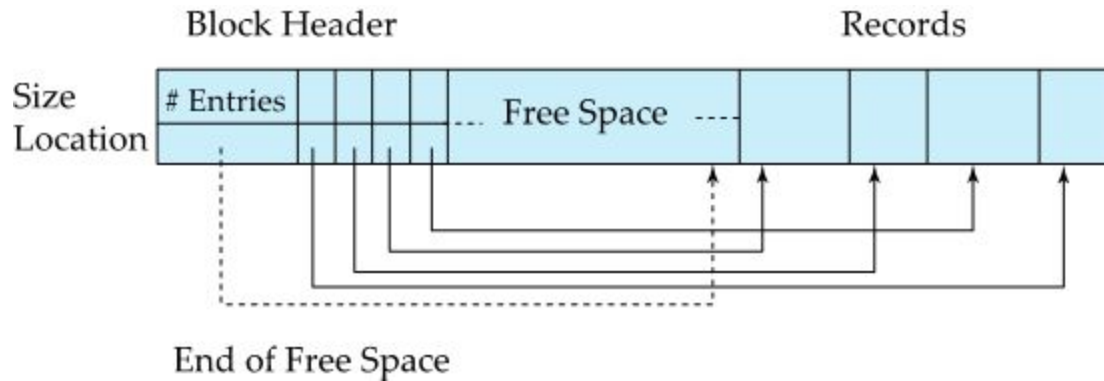
- To address storing variable-length records in a block
- Used to organize records within a block
- The header is at the beginning of each block and contains:
  - Number of record entries
  - End of free space in the block
  - An array whose entries are the location and size of each record
- The actual records are allocated contiguously in the block starting from the end of the block
- Insertion
  - Space is allocated for it at the end of free space, and an entry containing its size and location is added to the header
- Deletion
  - The space it occupies is freed and its entry is set to deleted (e.g., its size is set to -1)
  - The records in the block before the deleted record are moved so that the free space gets occupied with new insertions
- Records can grow or shrink in a similar fashion



- The cost of moving records within a block is not too high since the size of the block is limited
  - Remember, a block is the minimum read/write unit, anyway

#### Indirect Access to Slotted Page Structure

- No pointers point directly to a record
- Pointers must point to the entry in the header for that record
- This allows records to be moved around with insertions and deletions while hiding this movement from the application



#### Storing Large Objects

- Remember blob and clob types?

Blob	Clob
The full form of Blob is Binary Large Object.	The full form of Clob is Character Large Object.
This is used to store large binary data.	This is used to store large textual data.
This stores values in the form of binary streams.	This stores values in the form of character streams.
Using this you can store files like text files, PDF documents, word documents etc.	Using this you can stores files like videos, images, gifs, and audio files.

- Many databases restrict the size of a record to be no larger than the size of a block
- Records can logically contain large objects, but they are internally stored separately from other short attributes
- A logical pointer to the object is stored in the record
- Alternatives:
  - Store as files managed by the database
  - Store as files in file systems

## Organization of Records in Files

- Given a set of records, how to organize them in a file?
- Heap
  - Record can be placed anywhere in the file where there is space
- Sequential
  - Store records in sequential order, based on the value of the search key of each record
- Multitable clustering
  - Records of several different relations can be stored in the same file
  - Helpful when joining two relations if the two records to be joined are on the same block
- B+-tree
  - Ordered storage even with inserts/deletes
- Hashing
  - A hash function is computed on a search key such that all records that have the same hash value should be stored on the same block

## Heap File Organization

- Records can be placed anywhere in the file where there is free space
- In such organization, it is important to efficiently find blocks with free space within a file without having to sequentially search through all blocks of the file
- Free-space map keeps track of which blocks have free space to store records

## Free-Space Map

- An array containing 1 entry for each block in the relation
- Each entry is a few bits to 1 byte in size
- Stores a fraction  $f$  such that at least a fraction  $f$  of the space in the block is free

4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Example:
  - 16 blocks and 3 bits per block
  - Value is divided by 8 indicating the fraction of block that is free
  - When a record is inserted, the map is checked to find a block with free space to store the record. If none exist, a new block is allocated
- This map is stored in a file that is fetched into memory for faster access
- Whenever a record is inserted, updated, or there's a change in size that affects the fractions in the map, it is updated
- Scanning the free-space map is much faster than scanning all blocks to find free space
- However, it can still be very slow for large files
- Create a second-level free-space map
- Store the maximum of value among  $n$  entries in the main free-space map
- Example:



4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

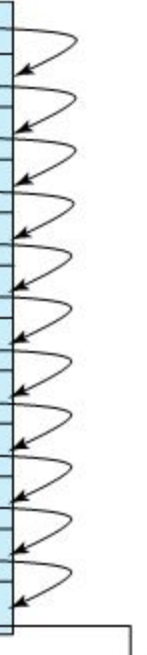
4	7	2	6
---	---	---	---

- More levels can be created

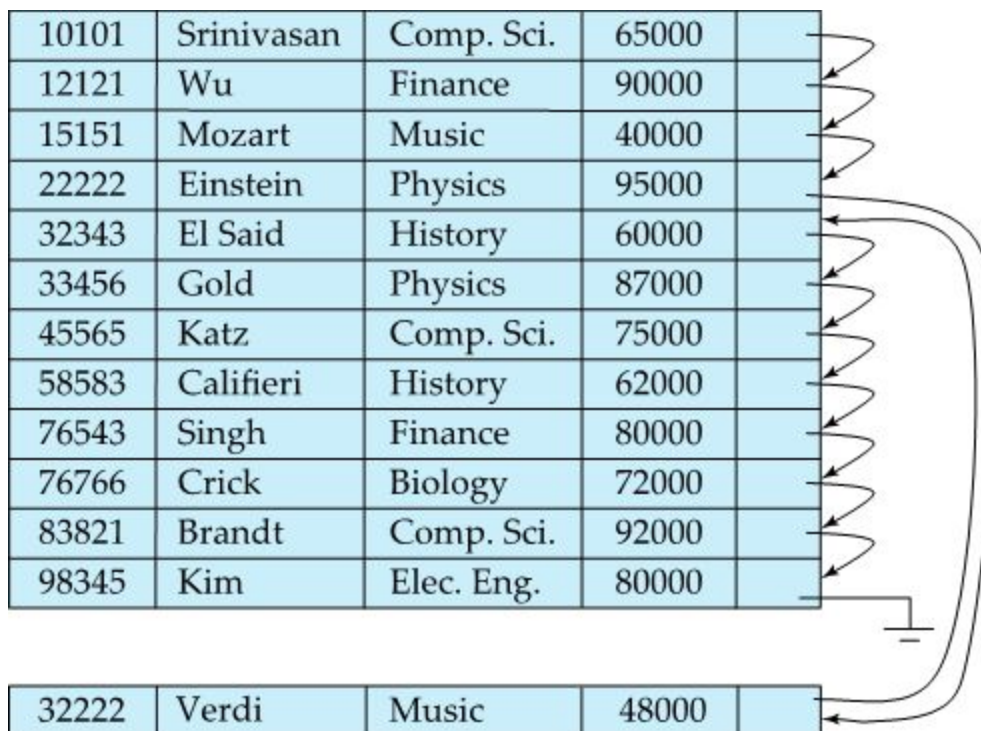
## Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key
- Search-key doesn't need to be a primary key, candidate key, or superkey

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



- Maintaining sequential order is achieved using pointers
- Deletion: Using pointer chains as we've seen earlier
- Insertion: Locate the position where the record is to be inserted
  - If there's free space, insert there
  - If no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
  - After a lot of insertions, the correspondence between physical order and search-key order may be lost
    - Periodical reorganization needs to happen during times when the system load is low



#### Multitable Clustering File Organization

- Store several relations in one file
  - Department

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

- Instructor

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

- Multitable clustering of department and instructor

Comp. Sci.	Taylor	100000	
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
Physics	Watson	70000	
33456	Gold	Physics	87000

- 
- Why?
- Consider this query:
 

```
select dept_name, building, budget, ID, name, salary
from department natural join instructor
```
- Slow processing for other types of queries
 

```
select *
from department
```
- Cluster key is the attribute that defines which records are stored together

## Partitioning

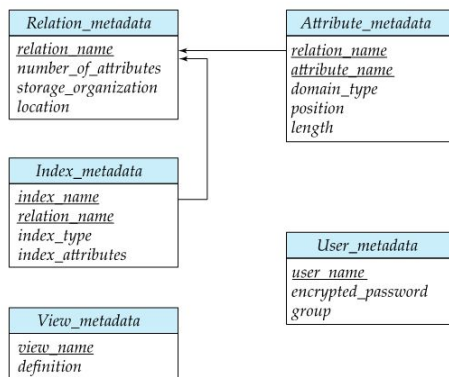
- Records in a relation can be partitioned into smaller relations that are stored separately
- Assume a transactions relation with a year attribute that represents the year of a transaction
  - transactions can be partitioned into smaller relations corresponding to each year. i.e., transactions\_2019, transactions\_2018, etc.
  - select \*
  - from transaction
  - where year=2019
- Queries written on transactions are issued over all relations unless a condition year = <year> exist
- Partitioning reduces the overhead of free space management for larger relations
  - Allows using different storage devices
- transactions\_2019 can be stored on SSD, while transaction\_2018 can be stored on magnetic disk

## DATA DICTIONARY STORAGE

- Data dictionary (system catalog) stores metadata about the relations in the database
  - Names of relations
  - Names of attributes of each relation
  - Domains and length of attributes
  - Names of views defined on the database, and their definition
  - Integrity constraints
- Stores information on users

- Names of users, their passwords, and the default schemas of users
  - Information on the authorizations for each user
- Stores information on storage organization
  - How a relation is stored (sequential/heap/...)
  - Physical location of a relation
- Statistical and descriptive data
  - Number of tuples in each relation
  - Number of distinct values for each attribute
- Information on indices

## REPRESENTATION OF METADATA



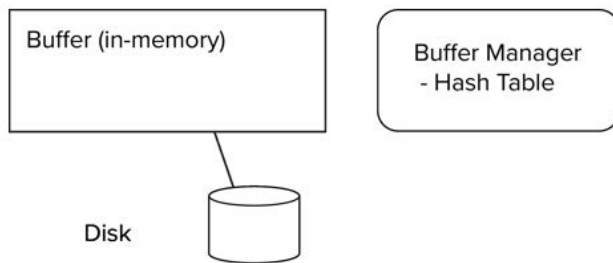
- Also represented using the relational model
  - Simplifies the overall structure of the system
  - Harness full power of the database for fast access of system data
- Whenever the database system needs to retrieve records from a relation, it must first access the catalog to find the location and storage organization of the relation
- How about the metadata of the catalog? :D
  - In the database code
  - Or in a fixed location in the database

## STORAGE ACCESS

- Even with the large increase in the size of main memory, most databases reside on disk
- Data must be brought to memory to be read or updated
- Updated blocks must be written back to disk
- A major goal of database systems is to minimize the number of block transfers between disk and memory
- One way of achieving this goal is by keeping as many blocks as possible in main memory so that when a block is accessed, it is already in main memory
- Need to be clever about which blocks to store in memory

## DATABASE BUFFER

- Space available in main memory to store copies of disk blocks
- The subsystem responsible for allocating buffer space is called the buffer manager



## BUFFER MANAGER

- When the database system needs to read a block from disk, a request is made to the buffer manager
- Block is already in the buffer
  - The address of the block in main memory is sent to the requester
- Block is not in the buffer
  - Space is allocated for the new block
    - If there is no free space in the buffer, some existing blocks are evicted
    - Evicted blocks are written back to disk if they were updated earlier
- The block is read from disk and stored in the allocated space
- The address of the allocated space is sent to the requester

## PINNED BLOCKS (LITERALLY A SEMAPHORE FROM 3000)

- When a block is in the buffer and being read by a database process, it may be evicted by the buffer manager if another process requests a read for a block that is not already in the buffer
  - The reader process will read incorrect data
  - If the first process is writing data, it will corrupt the data in the replacement block
- A block can be pinned in the buffer if a process is reading from it
- Buffer manager cannot evict a pinned block
- When the process is done reading, it will unpin the block
- Database systems should carefully manage pinned blocks to avoid pinning too many blocks
- If too many blocks are pinned, the buffer will be filled up quickly and the database will not carry out any further processing
- Multiple processes can read data from a block in the buffer
  - Each is required to execute a pin operation before accessing data
  - Unpin after finishing
- A block cannot be evicted until all processes unpin it
- Can be maintained using a pin count

## SHARED AND EXCLUSIVE LOCKS

- A process that adds or deletes a tuple from a page may need to move the page contents (as discussed in last lecture)
- During this period, no other process should read the contents of the page
- A database process can lock a buffer (shared lock or exclusive lock)
- Release the lock on the page after the process is finished reading or writing

- Shared locks for reading and exclusive locks for writing

## RULES OF LOCKING

- Any number of processes may have shared locks on a block at the same time
- Only one process is allowed to get an exclusive lock at a time
  - When a process has an exclusive lock, no other process may have a shared lock on the block
  - Exclusive locks can be granted only when no other process has a lock on a page
- If a process requests a shared lock when a block is not locked or already shared-locked, the lock is granted
- If another process has an exclusive lock, the shared lock is granted only after the exclusive lock is released

## ACQUIRING AND RELEASING LOCKS

- Before carrying out an operation on a block, a process must pin the block
  - Locks are obtained and must be released before unpinning the block
- Before reading data from a buffer, a process must get a shared lock on the block
  - When it's done reading, the lock must be released
- Before updating the contents of a block, a process must get an exclusive lock on the block
  - When it's done updating, the lock must be released

## BUFFER REPLACEMENT STRATEGIES

- The goal of a replacement strategy is to minimize accesses to the disk
- For general-purpose programs, it is difficult to predict which blocks will be read next
  - Operating systems use the past pattern of block references to predict future references
  - The assumption generally is that a block that has been referenced recently will be referenced again in the near future
  - If a block must be replaced, the least recently referenced block is evicted
  - Least recently used (LRU) block replacement scheme
- A database system can predict the pattern of future references more accurately

```
select *
from instructor natural join department;
```

•



```

for each tuple i of instructor do
  for each tuple d of department do
    if i[dept_name] = d[dept_name]
    then begin
      let x be a tuple defined as follows:
      x[ID] := i[ID]
      x[dept_name] := i[dept_name]
      x[name] := i[name]
      x[salary] := i[salary]
      x[building] := d[building]
      x[budget] := d[budget]
      include tuple x as part of result of instructor ⋈ department
    end
  end
end
end

```

- 
- Assume that the two relations are stored on different files
- Once a tuple of instructor is processed, it is not needed again
- So, once processing of an entire block of instructor tuples is completed, the block is no longer needed in main memory and can be evicted although it is recently used
- Toss-immediate strategy
- For the department tuples, every block needs to be examined once for each tuple of the instructor relation
- When processing of a block is completed, we know the block will not be accessed again until all other department blocks are processed
- The most recently used department block is the final block to be re-referenced
- The least recently used department block is the block that will be referenced next
- Most recently used (MRU) strategy
- Use statistical information about the probability that a request will reference a particular relation
  - For example, the catalog is one of the most frequently accessed parts of the database
  - The buffer manager should try not to evict catalog blocks

## COLUMN-ORIENTED STORAGE

- So far, we have been seeing data stored in a file with all values of attributes in a record stored together
  - Row-oriented storage
- In contrast, in column-oriented storage, values of each attribute of a relation are stored separately

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- In its simplest form, each column is stored in a separate file
- If a query needs to access the entire contents of the *i*th row of a table, the values at the *i*th position in each of the columns is retrieved and used to reconstruct the row
- Fetching multiple attributes of a single tuple requires multiple I/O operations
- Suitable for data analysis queries, where many rows are processed, but often only access some of the attributes

## BENEFITS OF COLUMN-ORIENTED STORAGE

- Reduced I/O
  - When a query accesses only a few attributes of a relation, the remaining attributes are not fetched from disk into memory
  - In contrast, in row-oriented storage, the unneeded attributes are also fetched
- Improved CPU cache performance
  - The adjacent bytes that are brought to cache contain data that will be accessed next
- Improved compression
  - Storing values of the same type together significantly increases the effectiveness of compression
  - Reduce the time taken to retrieve data from disk
- Vector processing
  - Allows a CPU operation to be applied in parallel on a number of elements of an array
  - Storing data column-wise allows vector processing of operations (e.g., comparing an attribute value with a constant)

## DRAWBACKS OF COLUMN-ORIENTED STORAGE

- Cost of tuple reconstruction



- Requires more I/O operations
- Cost of tuple deletion and update
  - In addition to higher I/O operations, deleting or updating in a compressed representation would require rewriting the entire sequence of tuples that are compressed in one unit
- Cost of decompression
  - Fetching data from compressed representation requires decompression

## HYBRID ROW/COLUMN STORES

- Some databases (e.g., SAP HANA) support the two underlying storage systems
- The row-oriented is designed for transaction processing
- The column-oriented is designed for data analysis
- Tuples are normally created in the row-oriented store
- Later migrated to the column-oriented store when they are no longer likely to be accessed in row-oriented manner

## STORAGE ORGANIZATION IN MAIN-MEMORY DATABASES

- What if we have enough memory in which the whole database can fit?
- One way to utilize this is to allocate a large amount of memory to the database buffer so that the entire database will reside in the buffer and disk I/O will be avoided
  - Updated blocks still have to be written back to disk
- Performance can improve significantly by tailoring the storage organization and database data structures to exploit the fact that the database can fit in memory

## EXAMPLE

- Consider the cost of accessing a record given a record pointer
- Records are stored in blocks
- Pointers to records consist of a block identifier and an offset or slot number within the block
- Steps
  - Check if the block is in the buffer (by checking an in-memory Hash index)
  - Find where in the buffer it is located (through the buffer manager)
    - If not in the buffer, wait until it is fetched
- Consumes a significant number of CPU cycles
- We can exploit the fact that the database is already in memory by keeping direct pointers to records in memory
- As long as records are not moved around!
  - Loading from buffer and eviction from buffer is no longer an issue
- If records are stored in a slotted-page structure within a block, records may move within a block with insertions and deletions
  - Direct pointers to records are not possible in this case
  - Locking of the block may be required to ensure that a record does not get moved while another process is reading its data
- Many main-memory databases do not use a slotted-page structure for allocating records
- They directly allocate records in main-memory
  - Memory may get fragmented after a while

- Periodically perform compaction of memory

## Database structures

### BASIC CONCEPTS

- Indexing mechanisms are used to speed up access to desired data
- Without indices, every query would end up reading the entire contents of every relation that it uses
- Search key: Attribute (or set of attributes) that are used to look up records in a file
- Index entry: Consists of the pair (search-key, pointer)
  - search-key contains the value of the search-key attribute
  - pointer points to one or more records with the value stored in search-key
- Index file: The file that consists of records of index entries
  - For example, an index on student with a search key ID would contain records containing sorted values of ID's and pointers to where the records with these ID's are stored
- The index file is typically much smaller than the original file
- In previous example, keeping a sorted list of students' ID would not work well on very large databases
  - The index itself would be very large
  - Even though the index reduces search time, the process of finding a student can still be time-consuming due to the overhead of searching in the index
  - Updating a sorted list as students are added or removed can be also expensive

### INDEX EVALUATION

- Access types: The types of access that are supported efficiently
  - Finding records with a specified attribute value
  - Finding records whose attribute values fall within a range of values
- Access time: The time it takes to find a particular data item
- Insertion time: The time it takes to insert a new data item
  - Time to find the correct place to insert
  - Time to update the index structure
- Deletion time: The time it takes to delete a data item
  - Same as insertion
- Space overhead: The additional space occupied by the index

### KINDS OF INDICES

- Ordered indices: Based on a sorted ordering of the values
- Hash indices: Based on a uniform distribution of values across a range of buckets using a hash function

## ORDERED INDICES

- Stores the values of the search keys in sorted order and associates with each search key the records that contain it
- A relation may have several indices on different search keys
- Clustering index (primary index): The index whose search key also defines the sequential order of the file
- Non-clustering index (secondary index): The index whose search key specifies an order different from the sequential order of the file
- There are two types of ordered indices:
  - Dense index: One index record appears for every search-key value
  - Sparse index: An index entry appears for only some of the search-key values

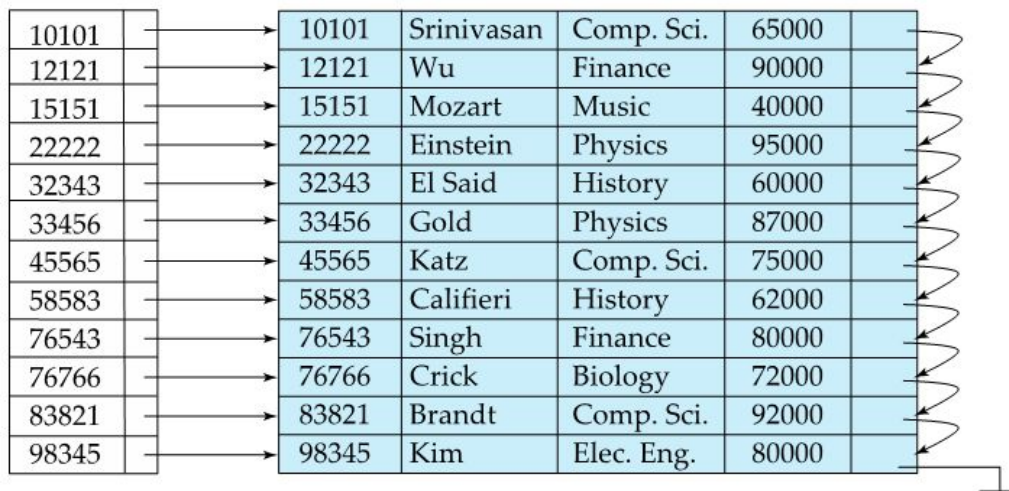
## DENSE INDEX

- One index record appears for every search-key value
- Primary index: The index record contains the search-key value and points to the first data record with that value
  - The rest of records with the same search-key value would be stored sequentially after the first record
- Secondary index: The index must store a list of pointers to all records with the same search-key value

## EXAMPLE OF DENSE INDEX (1)

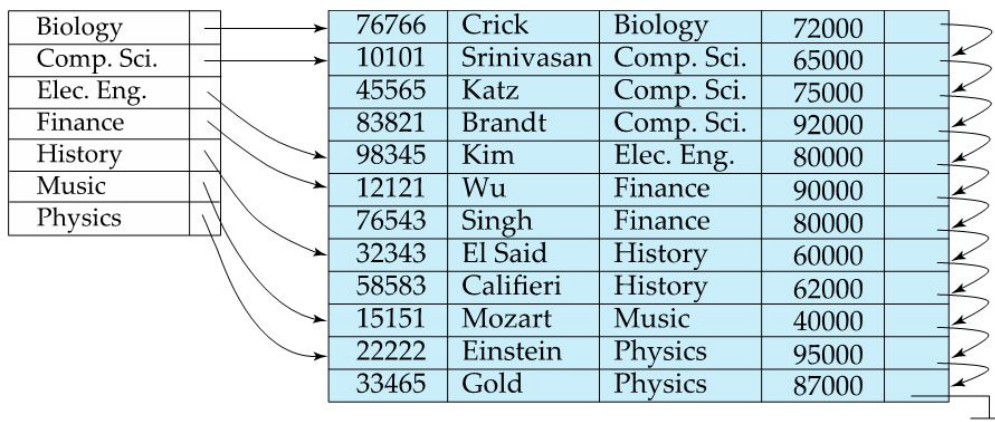
- Example: Index on ID (primary key) attribute of instructor relation

10101	→	10101	Srinivasan	Comp. Sci.	65000	
12121	→	12121	Wu	Finance	90000	
15151	→	15151	Mozart	Music	40000	
22222	→	22222	Einstein	Physics	95000	
32343	→	32343	El Said	History	60000	
33456	→	33456	Gold	Physics	87000	
45565	→	45565	Katz	Comp. Sci.	75000	
58583	→	58583	Califieri	History	62000	
76543	→	76543	Singh	Finance	80000	
76766	→	76766	Crick	Biology	72000	
83821	→	83821	Brandt	Comp. Sci.	92000	
98345	→	98345	Kim	Elec. Eng.	80000	



## EXAMPLE OF DENSE INDEX (2)

- Dense index on dept\_name with the instructor file sorted on dept\_name

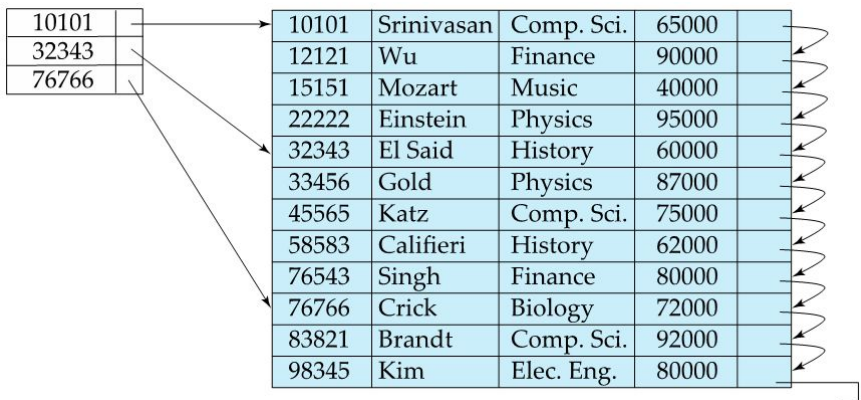


## SPARSE INDEX

- An index entry appears for only some of the search-key values
- Can be used only if the relation is stored in sorted order of the search-key (primary index)
- The index record contains the search-key value and points to the first data record with that value
- To locate a record:
  - Find the index entry with the largest search-key value that is less than or equal to the search-key value we are looking for
  - Starting at the record pointed to by the index entry, we sequentially scan following records until the search-key value we are looking for is found

## EXAMPLE OF SPARSE INDEX

- Find the instructor with ID = 45565



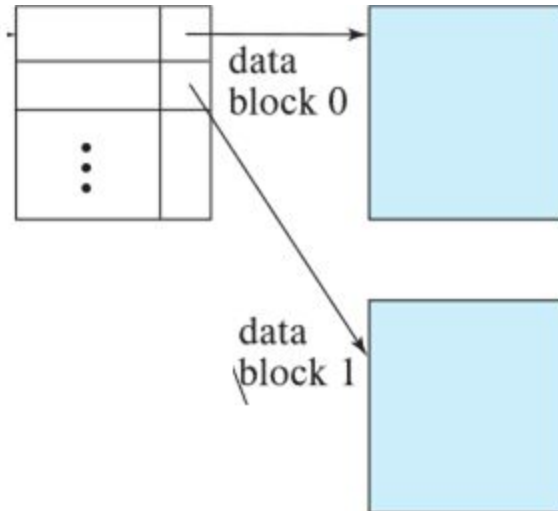
## DENSE VS. SPARSE INDEX

- Dense index
  - Faster to locate a record
  - Takes more space
  - More maintenance overhead for insertion and deletion
- Sparse index
  - Slower to locate a record
  - Takes less space

- Less maintenance overhead for insertion and deletion

## GOOD TRADEOFF

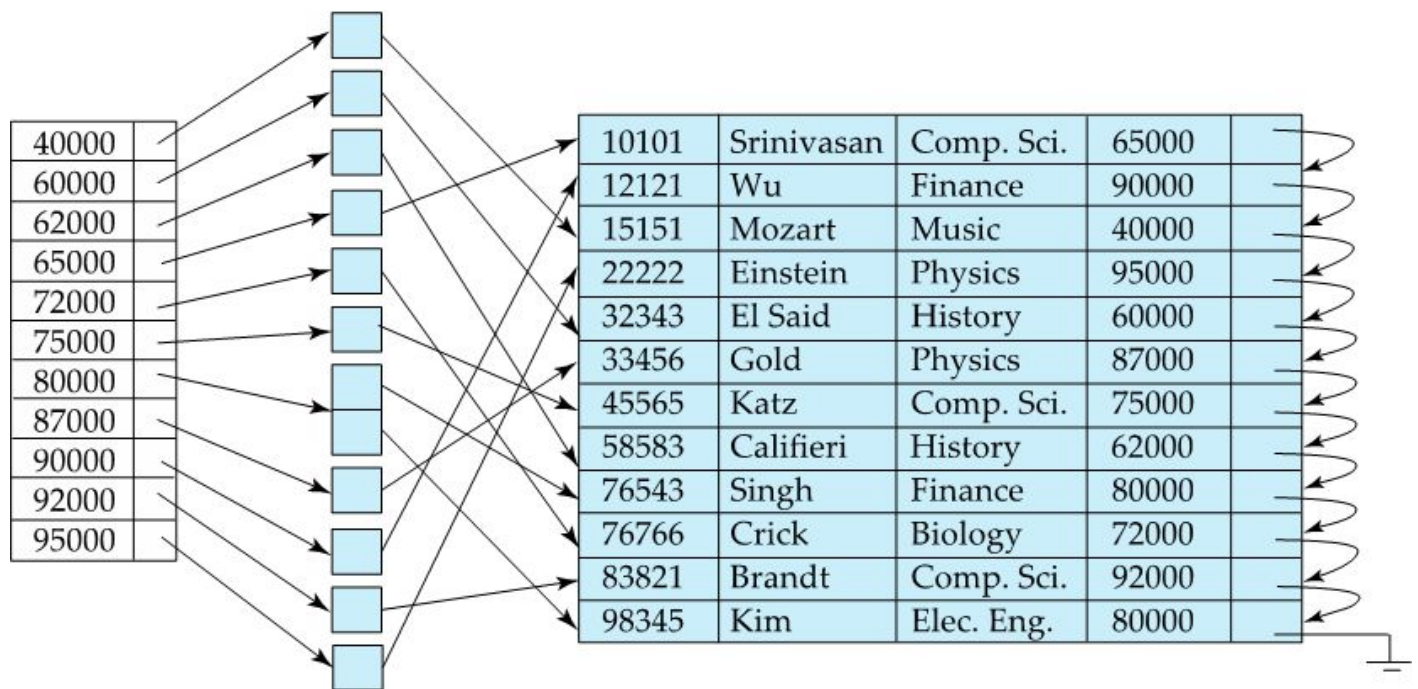
- For primary index, use sparse index with an index entry for every block in file, corresponding to least search-key value in the block



- 
- For secondary index, use sparse index on top of dense index (multilevel index)

## SECONDARY INDEX

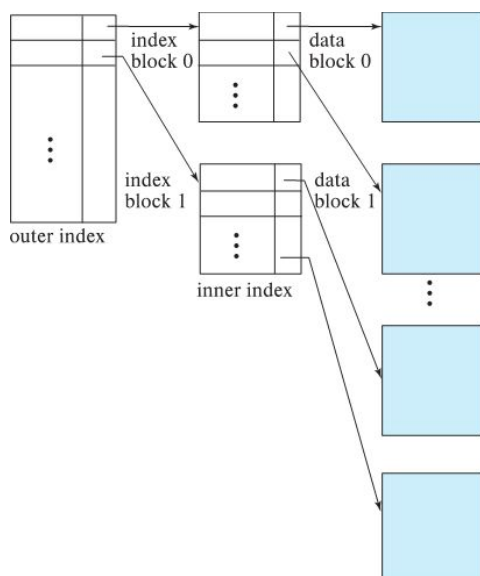
- Secondary indices must be dense (one index entry for every search-key value and pointer to every record in file)
- Secondary key on a candidate key looks just like a dense primary index except that the records pointed to by successive values in the index are not stored sequentially
- If a relation can have more than one record containing the same search-key value, the search key is said to be a nonunique search key
- The pointers in the secondary index do not point directly to the records, but to buckets that contain the pointers to records



## CASE FOR MULTILEVEL INDICES

- Suppose that each 100 index entries fit on a 4K block
- Suppose we build a dense index on a candidate key of a relation with 1,000,000 records
- The index will occupy 10,000 blocks or 40 MB
- If 100,000,000 tuples, the index will occupy 4 GB
- If the index is small enough to be kept in memory, search time is low
- If index is too large, index blocks must be fetched from disk
- If we look up an entry using binary search, we would require as many as  $\log(b)$  block reads, where  $b$  is the number of blocks of the index
  - Note that the blocks are not adjacent in this case, so each block read requires a random access I/O operation

## MULTILEVEL INDEX(PAGE LEVELS IN 3000)



- Construct a sparse outer index on the original index
- The original index is called the inner index
- Recall that index entries are always sorted in order, which allows the existence of a sparse outer index
- To locate a record:
  - Use binary search on the outer index to find the record for the largest search-key value less than or equal the value we are looking for
  - The pointer points to a block of the inner index
  - Scan the block (or binary search) until we find the record that has the largest search-key value less than or equal the value we are looking for

- The pointer in this record points to the block of the file that contains the record we are looking for

## MULTILEVEL INDEX

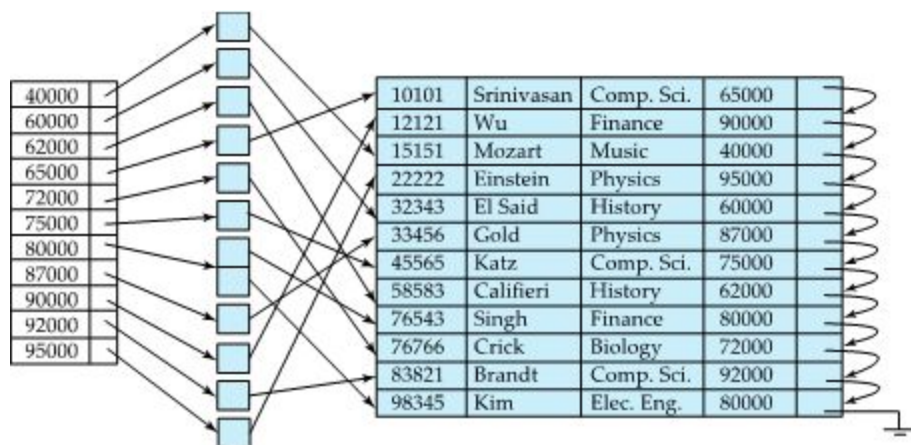
- The outer index can fit in main memory for faster search
  - This will result in reading only one block from disk (for the inner index) before we finally read the block of the record we are looking for
  - Compare this to the  $\log(b)$  reads required for one level of indexing
- If the outer index is also too large to fit in main memory, another level of index can be created
- Indices at all levels must be updated on insertion or deletion
- Regardless of what form of index is used, every index must be updated whenever a record is inserted, updated, or deleted
- 

## UPDATING THE INDEX (INSERTION)

- Assume single-level indexing for now
- Perform a lookup using the search-key value of the record to be inserted
- Dense Indices
  - If the search-key value does not appear in the index, insert an index entry with the search-key value in the index at the appropriate position
    - Indices are maintained as sequential files, so we need to create space for new entry (use an empty space in the file or use the overflow block)
  - Otherwise, if the search-key value appears in index:
    - If the index entry stores pointers to all records with the same search-key value, add a pointer to the new record in the index entry
    - Otherwise, the index entry stores a pointer to only the first record with the search-key value. So, place the record being inserted after the other records with the same search-key value

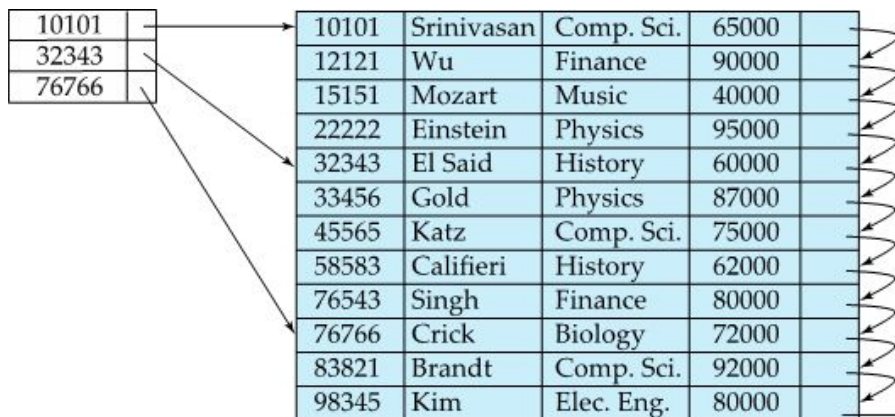
10101	→	10101	Srinivasan	Comp. Sci.	65000	→
12121	→	12121	Wu	Finance	90000	→
15151	→	15151	Mozart	Music	40000	→
22222	→	22222	Einstein	Physics	95000	→
32343	→	32343	El Said	History	60000	→
33456	→	33456	Gold	Physics	87000	→
45565	→	45565	Katz	Comp. Sci.	75000	→
58583	→	58583	Califieri	History	62000	→
76543	→	76543	Singh	Finance	80000	→
76766	→	76766	Crick	Biology	72000	→
83821	→	83821	Brandt	Comp. Sci.	92000	→
98345	→	98345	Kim	Elec. Eng.	80000	→





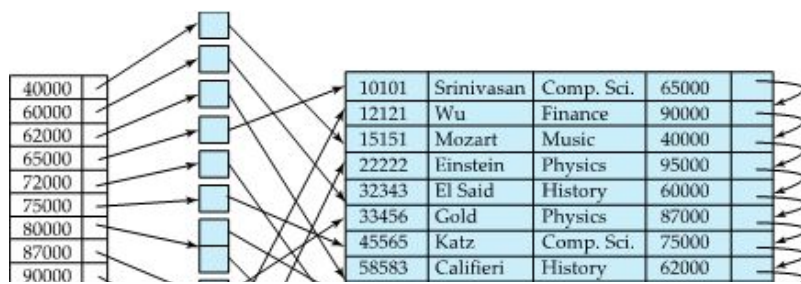
- Sparse Indices

- We assume that the index stores an entry for each block
- If the system creates a new block for the new record, it inserts the first search-key value appearing in the new block into the index
- If the new record has the least search-key value in its block, update the index entry pointing to the block
- If not, no changes are made to the index



## UPDATING THE INDEX (DELETION)

- The system first looks up the record to be deleted
- Dense Indices
- If the deleted record was the only record with its search-key value, delete the corresponding index entry from the index
- Otherwise:





- If the index entry stores pointers to all records with the same search-key value, delete the pointer to the deleted record from the index entry
- Otherwise, the index entry stores a pointer to only the first record with the same search-key value, if the deleted record was the first record with the search-key value, update the index entry to point to the next record

- Sparse Indices

- If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index

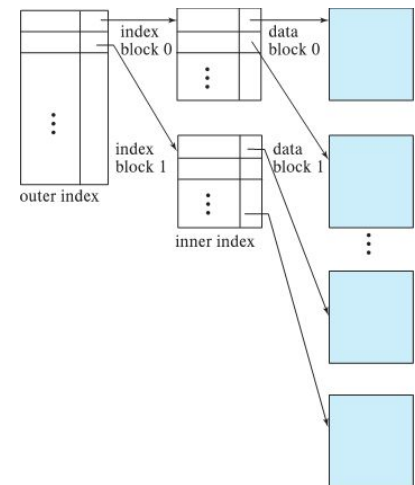
- Otherwise:

- If the deleted record was the only record with its search key, replace the corresponding index record with an index record for the next search-key value. If the next search-key value already has an index entry, the entry is deleted
- Otherwise, if the index entry for the search-key value points to the record being deleted, update the index entry to point to the next record with the same search-key value

10101	10101	Srinivasan	Comp. Sci.	65000
32343	12121	Wu	Finance	90000
76766	15151	Mozart	Music	40000
	22222	Einstein	Physics	95000
	32343	El Said	History	60000
	33456	Gold	Physics	87000
	45565	Katz	Comp. Sci.	75000
	58583	Califieri	History	62000
	76543	Singh	Finance	80000
	76766	Crick	Biology	72000
	83821	Brandt	Comp. Sci.	92000
	98345	Kim	Elec. Eng.	80000

## INSERTION AND DELETION FOR MULTILEVEL INDICES

- Simple extension of what we previously discussed
- The system updates the lowest level index first
- For the second level index, the lowest-level index is considered a file with records
  - If there is any change in the lowest-level index, the system updates the second-level index in the same way described before
- More levels are updated in the same manner



## INDICES ON MULTIPLE KEYS

- Search key can have more than one attribute (composite search key)
- The search key can be represented as  $(a_1, a_2, \dots, a_n)$ , where the indexed attributes are  $A_1, A_2, \dots, A_n$
- The ordering of the search-key values is the lexicographic ordering
  - For example,  $(a_1, a_2) < (b_1, b_2)$  if either  $a_1 < b_1$  or  $a_1 = b_1$  and  $a_2 < b_2$
- An index on takes on the composite search key (course\_id, semester, year) would be useful to find all students who have registered for a particular course in a particular semester/year

## B+ Tree

### DISADVANTAGES OF INDEX-SEQUENTIAL FILE ORGANIZATION

- When relations grow, the performance degrades
- Index files grow and many overflow blocks get created
- Requires periodic reorganization of the entire index

### B+-TREE INDEX

- Most widely used of several index structures that maintain their efficiency despite insertion and deletion of data
- B+-tree is considered a multilevel index, but has a different structure
- Balanced tree
  - Every path from the root to a leaf is of the same length
- Automatically reorganizes itself
- Reorganization of the entire file is not required to maintain performance
- More overhead for insertions and deletions
- Advantages of B+-trees outweigh disadvantages

### B+-TREE NODE STRUCTURE

- For now, assume that there are no duplicate search-key values
- Typical node

$P_1$	$K_1$	$P_2$	$\dots$	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	---------	-----------	-----------	-------

- $K_i$  are the search key values
- $P_i$  are pointers to children (for non-leaf nodes) or pointers to records (for leaf nodes)
- The search-keys in a node are ordered
$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

### LEAF NODES IN B+-TREES

- For  $i = 1, 2, \dots, n-1$ , pointer  $P_i$  points to a file record with search-key value  $K_i$
- $P_n$  points to next leaf node in search-key order
- If  $L_i, L_j$  are leaf nodes and  $i < j$ ,  $L_i$ 's search-key values are less than or equal to  $L_j$ 's search-key values

### NON-LEAF NODES IN B+-TREES

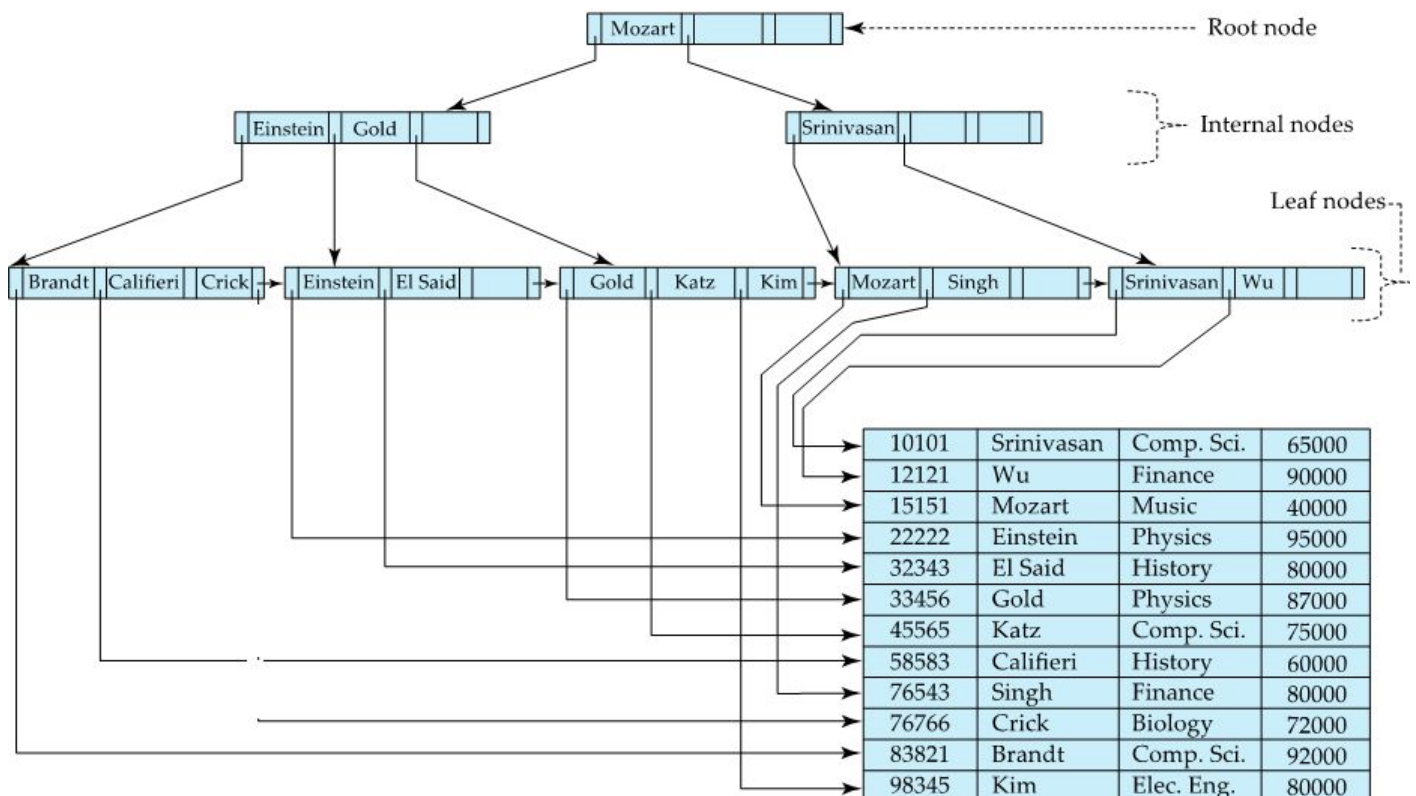
- Non-leaf nodes form a multi-level sparse index on the leaf nodes
- For a non-leaf node with  $m$  pointers ( $m \leq n$ ):
  - All the search-keys in the subtree to which  $P_1$  points are less than  $K_1$

- For  $2 \leq i \leq m-1$ , all the search-keys in the subtree to which  $P_i$  points have values greater than or equal to  $K_{i-1}$  and less than  $K_i$
- All the search-keys in the subtree to which  $P_m$  points have values greater than or equal to  $K_{m-1}$

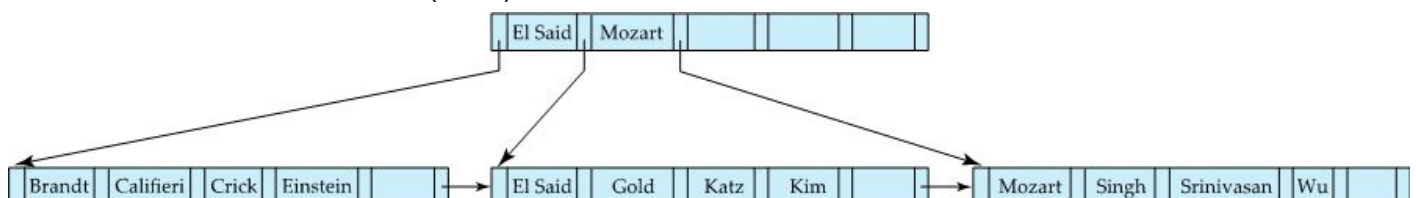


## B+-TREE STRUCTURE

- All paths from the root to leaf are of the same length
- Each node that is not a root or a leaf has between  $n/2$  and  $n$  pointers
- A leaf node has between  $(n-1)/2$  and  $n-1$  values
- Special cases for the root:
  - If the **root is not a leaf**, it has **at least 2 children**
  - If the **root is a leaf** (that is, there are no other nodes in the tree), it can have **between 0 and  $(n-1)$  values**
- EXAMPLE OF B+-TREE ( $n = 4$ )



- EXAMPLE OF B+-TREE ( $n = 6$ )



- Leaf nodes must have between 3 and 5 values ( $((n-1)/2$  and  $n-1$ , with  $n = 6$ ))

- Non-leaf nodes other than root must have between 3 and 6 pointers ( $n/2$  and  $n$  with  $n \geq 6$ )
- Root must have at least 2 children

## OBSERVATIONS ABOUT B+-TREES

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices
- The B+-tree contains a relatively small number of levels
  - Level below root has at least  $2 \cdot n/2$  values
  - Next level has at least  $2 \cdot n/2 \cdot n/2$  values
  - ...
- If there are  $K$  search-key values in the file, the tree height is no more than  $\log_{n/2}(K)$
- DUPLICATE SEARCH-KEY VALUES
- In general, search keys could have duplicates
- Option 1: Modify the tree structure to store each search key at a leaf node as many times as it appears in records
  - This approach can result in duplicate search-key values at internal nodes
  - Will make insertion and deletion more complicated and expensive
- Option 2: Store a set of record pointers with each search-key value (similar to secondary index implementation we saw in the latest lecture)
  - More complicated and can result in inefficient access (if the number of record pointers for a particular key is very large)

### OPTION 3

- Make search keys unique
- Suppose the desired search key attribute  $a_i$  of relation  $r$  is nonunique
- Let  $A_p$  be the primary key for  $r$
- The unique composite search key  $(a_i, A_p)$  is used instead of  $a_i$
- Can be any other candidate key
- For example, for instructor, if we want to create an index on name, we create an index on (name, id) since id is the primary key of instructor

## QUERIES ON B+-TREES

function find( $v$ )

$C = \text{root}$

  while ( $C$  is not a leaf node)

    Let  $i$  be least number s.t.  $V \leq K_i$ .

    if there is no such number  $i$  then

      Set  $C = \text{last non-null pointer in } C$

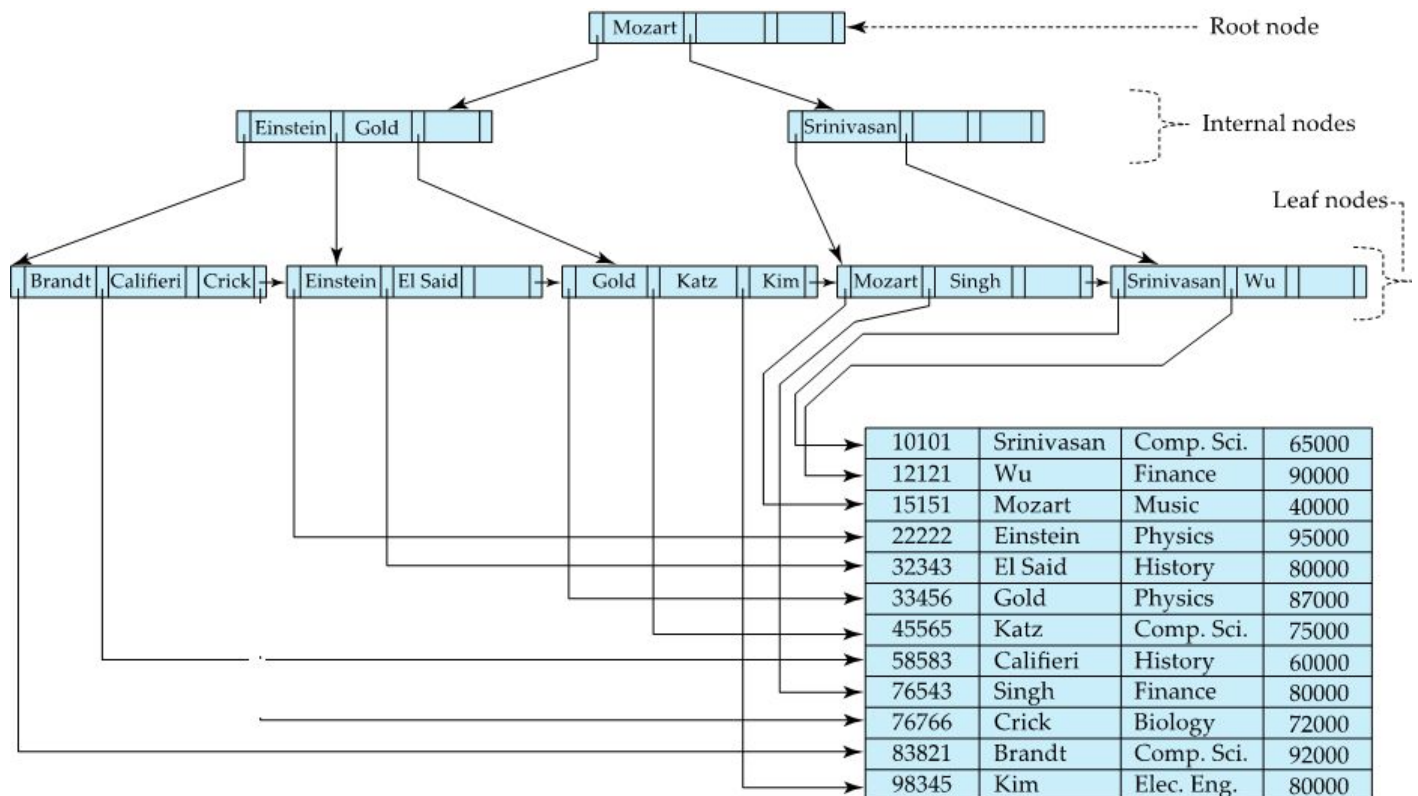
    else if ( $v = C.K_i$ ) Set  $C = P_i + 1$

    else set  $C = C.P_i$

  if for some  $i$ ,  $K_i = V$  then return  $C.P_i$

  else return null /\* no record with search-key value  $v$  exists. \*/

## QUERIES ON B+-TREES



## QUERIES ON B+-TREES (RANGE QUERIES)

- To find all records with search-key values in a given range [lb, ub]
- Traverse to a leaf in a manner similar to find(lb)
- The leaf may or may not contain the value lb
- Scan this node and subsequent leaf nodes collecting pointers to all records with key values C.Ki such that  $lb \leq C.Ki \leq ub$
- Stop when  $C.Ki > ub$  or there are no more keys in the tree (reached last leaf node)

## COST OF QUERYING B+-TREES

- Traversal
  - If there are N records in the table, the path from root to a leaf node is no longer than  $\log_{n/2}(N)$
- A node is generally the same size as the size of a disk block (typically 4 KB)
  - With a search-key size of 12 bytes and a disk-pointer size of 8 bytes, n is around 200
  - With a search-key size of 32 bytes and a disk-pointer size of 8 bytes, n is around 100 The typical case
- With 1 million search key values and  $n = 100$ 
  - At most  $\log_{50}(1,000,000) = 4$  nodes are accessed in a lookup traversal from root to leaf
  - Root node is heavily accessed and probably in the database buffer
- Compare this to the cost of traversing a binary tree (balanced)
  - $\log_2(1,000,000) = 20$

- Cost of accessing actual records
- Range queries
  - Additional cost of scanning consecutive leaf nodes
  - If  $M$  pointers are retrieved, at most  $M/(n/2)$  leaf nodes need to be accessed
  - Cost of accessing actual records
  - For secondary indices, each record could be on a different block, requiring  $M$  random I/O operations in the worst case
  - For primary indices, records are stored sequentially in files, resulting in much less I/O operations

## RANGE QUERIES FOR NON-UNIQUE KEYS

- Search for  $a_i = v$  can be implemented by a range search on composite key, with range  $(v, -\infty)$  to  $(v, +\infty)$
- Range queries on  $a_i$  can be handled by the range  $(lb, -\infty)$  to  $(ub, +\infty)$

## UPDATES ON B+-TREES

- Insertions and deletions are more complicated than lookup
  - Split a node that becomes too large as a result of insertion (more than  $n$  pointers)
  - Coalesce (combine) nodes if a node becomes too small (fewer than  $n/2$  pointers)
- Assume temporarily that nodes never become too large or too small
- Insertion
  - Using the same technique for lookup, find the leaf node in which the search-key value would appear
  - Insert an entry in the leaf node such that the search keys are still ordered
- Deletion
  - Using the same technique for lookup, find the leaf node containing the entry to be deleted
  - Remove the entry from the leaf node
  - All entries in the leaf node to the right of the deleted entry are shifted left by one position

## UPDATES ON B+-TREES (INSERTION)

- Assume the record was already added to the file
- Let  $v$  be the search-key value and  $pr$  be the pointer to the added record
- Find the leaf node in which the search-key value would appear (using the lookup algorithm)
- If there is room in the leaf node, insert  $(pr, v)$  into the leaf node
- If there is no room, this leaf node will need to be split and pointers need to be updated

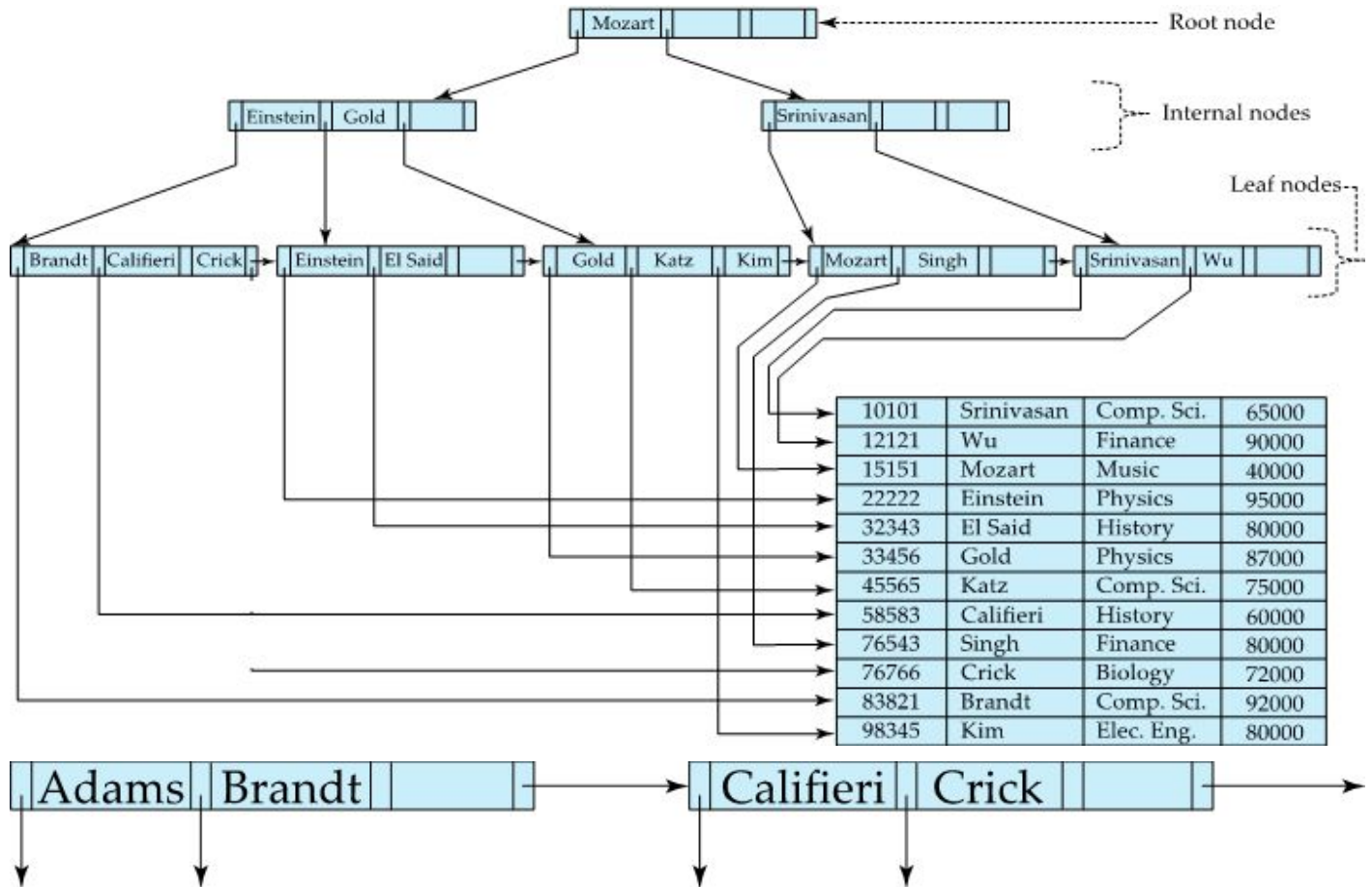
## SPLITTING A LEAF NODE

- Take the  $n$  (search-key value, pointer) pairs, including the one being inserted in sorted order
- Place the first  $n/2$  in the original node, and the rest in a new node
- Let the new node be  $p$ , and let  $k$  be the least key value in  $p$
- Insert  $(k, p)$  in the parent of the node being split
- If the parent is full, split it and propagate the split further up if needed
- In the worst case, the root will be split and a new root is created (height of tree is increased by 1)

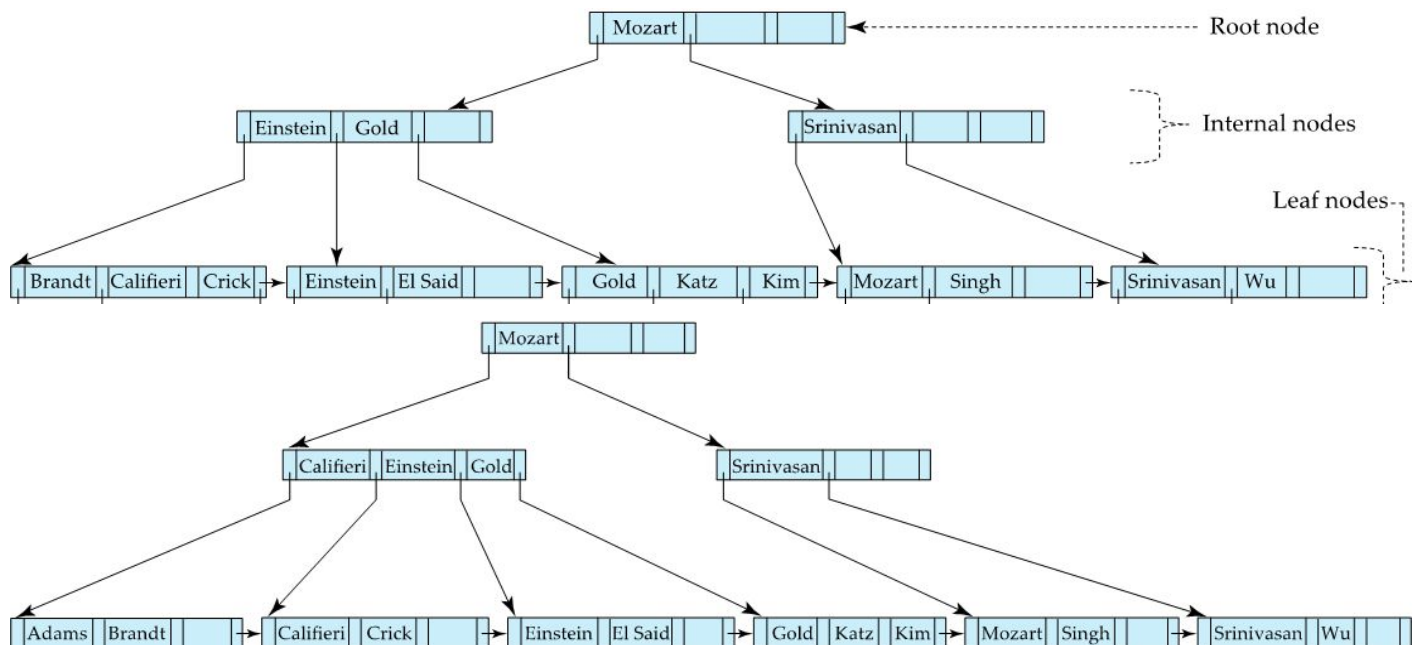


## SPLITTING A LEAF NODE (EXAMPLE)

- Insert an entry for “Adams” into:



- Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
- Next step: insert entry with (Califieri, pointer-to-new-node) into parent

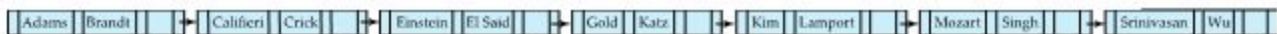
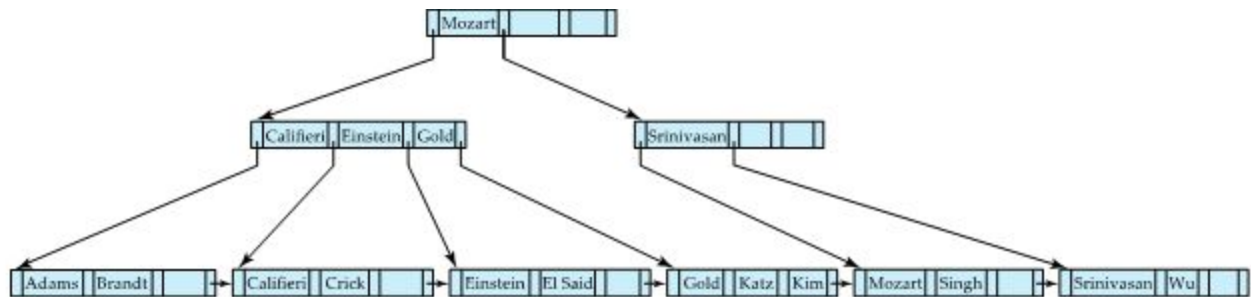


## SPLITTING A NON-LEAF NODE

- Inserting a new (k, p) into an already full internal node N
- Copy N to an in-memory area M with space for  $n+1$  pointers and  $n$  keys
- Insert (k, p) into M
- Copy  $P_1, K_1, \dots, K_{(n+1)/2-1}, P_{(n+1)/2}$  from M back into node N
- Copy  $P_{(n+1)/2+1}, K_{(n+1)/2+1}, \dots, K_n, P_{n+1}$  from M into newly allocated node N'
- Insert (N,  $K_{(n+1)/2}, N'$ ) into the parent of N and N'

#### SPLITTING A LEAF NODE (EXAMPLE) (CONT.)

- Insert another entry for "Lamport"



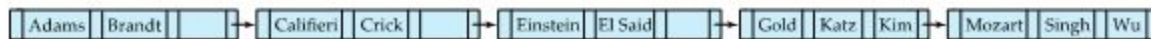
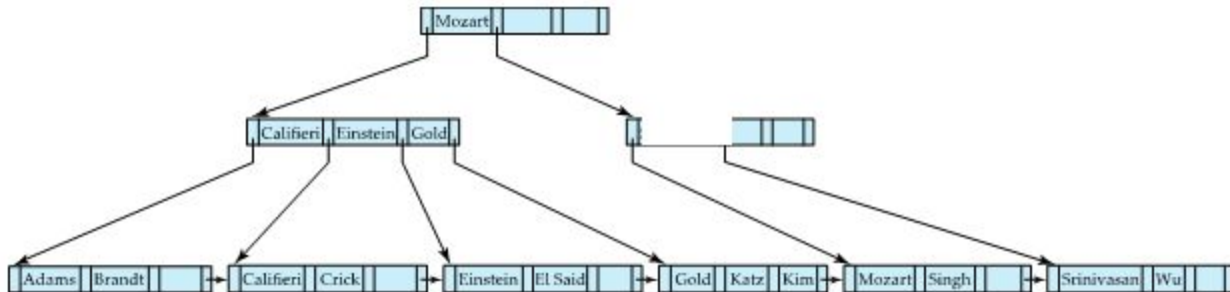
#### UPDATES ON B+-TREES (DELETION)

- Assume the record was already deleted from the file
- Let  $v$  be the search-key value of the record and  $pr$  be the pointer to the record
- Remove ( $pr, v$ ) from the leaf node
- If the node has too few entries due to the removal, and the entries in the node and sibling fit into a single node, then merge siblings:
  - Insert all the search-key values in the two nodes into a single node (the one to the left), and delete the other node
  - Delete the pair ( $K_{i-1}, P_i$ ), where  $P_i$  is the pointer to the deleted node, from its parent, recursively using the above procedure
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and sibling don't fit into a single node, then redistribute pointers:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries
  - Update the corresponding search-key value in the parent of the node
- The node deletions may cascade upwards until a node which has  $n/2$  or more pointers is found
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root



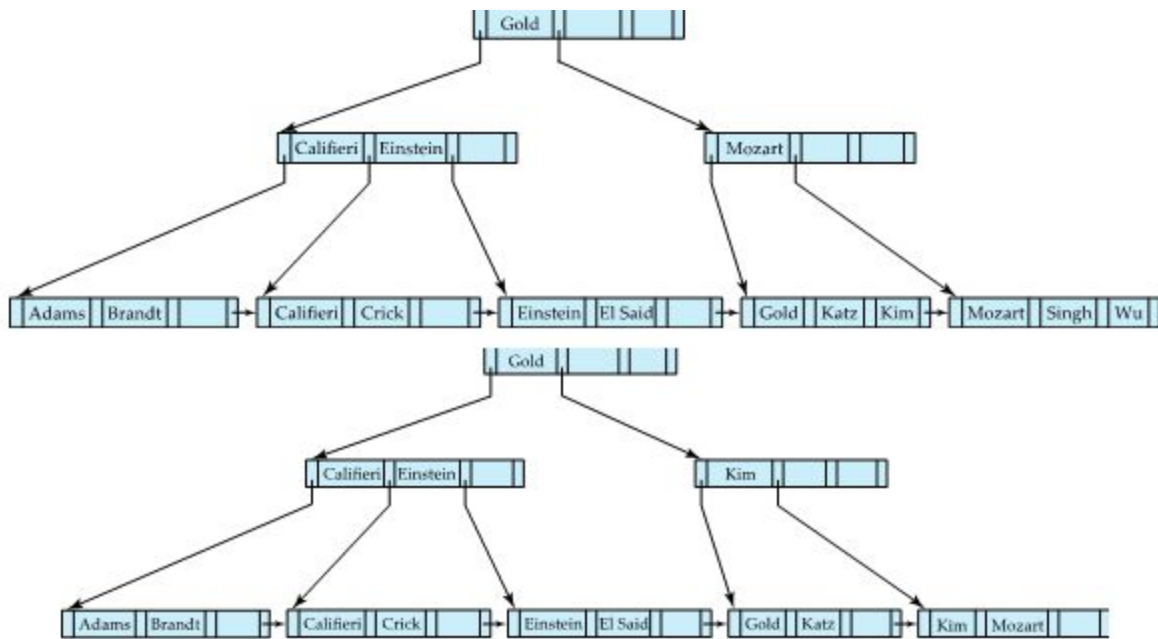
## DELETION EXAMPLE

- Delete the entry “Srinivasan”
- Delete the entry (Srinivasan, n) from parent
- Parent node has one pointer
- Sibling is full
- Redistribute pointers
- Move the rightmost pointer from sibling to the underfull node
- The underfull node would now have two pointers but no value separating them
- This value is in the next parent node (root)



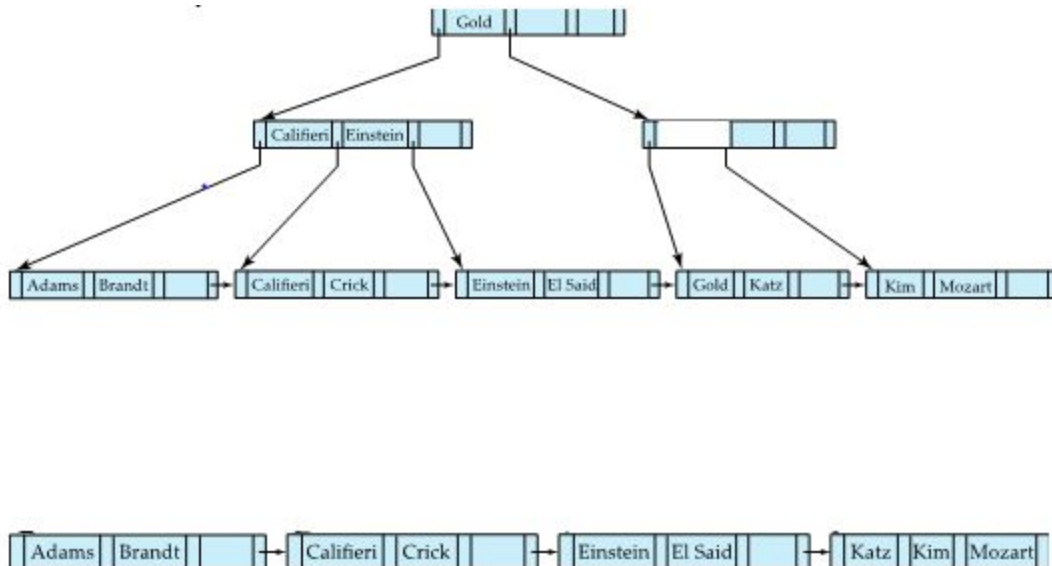
## DELETION EXAMPLE

- Delete the entries “Singh” and “Wu”



## DELETION EXAMPLE

- Delete the entry "Gold"



Lec 16 ends here

## Hash Indices

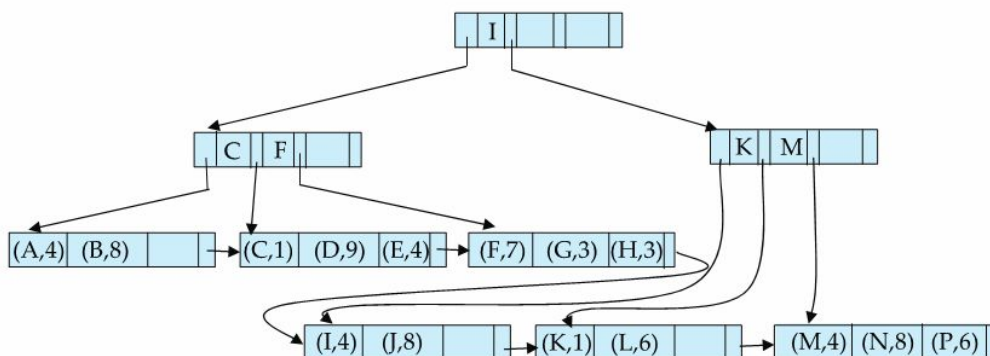
## COMPLEXITY OF UPDATING B+-TREES

- Although insertions and deletions are complicated, they require relatively few I/O operations
- Number of I/O operations of insertion and deletion of a single entry is proportional to  $\log_{2/2}(N)$ , where  $N$  is the number of records
  - Assuming no duplicate values for search key
- In other words, the I/O operations is proportional to the height of the B+-tree

- In practice, the number of I/O operations is less
- Most internal nodes tend to be in the database buffer
  - With fanout of 100, and assuming accesses to leaf nodes are uniformly distributed
  - The parent of a leaf node is 100 times more likely to get accessed than the leaf node
  - The number of non-leaf nodes will be a little more than 1/100th the number of leaf nodes
- Typically one or two I/O operations for lookups
- In practice, splits/merges are rare
  - Depending on the order of inserts, with fanout of 100, only from 1 in 100 to 1 in 50 insertions will cause a split
  - More than one block will be written
  - On average a little more than one I/O operation to write updated block
- Average node occupancy depends on insertion order
  - If entries are inserted in random order, nodes are expected to be more than two-thirds full on average
  - If entries are inserted in sorted order, nodes will be only half full

## B+-TREE FILE ORGANIZATION

- Recall that the main drawback of index-sequential file organization was when the file gets too large
  - Increasing percentage of both index entries and actual records get written in overflow blocks
- For the index, we use B+-tree
- For the actual records, it is also possible to use B+-tree to store actual records
- Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is usually less than the number of pointers in non-leaf node
- Insertions and deletions from a B+-tree file organization are handled in the same way as insertion and deletion in the B+-tree index



- Good space utilization is essential since records use more space than pointers
- Involve more sibling nodes in redistribution during splits and merges

## SECONDARY INDICES AND RECORD RELOCATION

- In the B+-tree file organization, the location of records change even if the records are not updated
  - When a node is split, some records may move to a new node
- What happens to secondary indices that store pointers to the relocated records?
  - Must be updated

- Each leaf node may contain a large number of records
- One split may require tens, or even hundreds of I/O operations to update all affected entries in the secondary index (these entries can be stored anywhere physically on disk)

## SECONDARY INDICES AND RECORD RELOCATION

- In the secondary index, store values of the primary-index search-key attributes instead of pointers to actual records
- Example: In the instructor relation, suppose we have a primary index on ID, and a secondary index on dept\_name
- The secondary index will store a list of instructor's ID values of the corresponding records instead of pointers to actual records
- Relocation of records does not require any change in the secondary index
- How to locate a record using the secondary index:
  - Use the secondary index to find the primary-index search-key value
  - Use the primary index to find the corresponding record
  - Increases the cost of accessing data, but reduces the cost of index update

## INDEXING STRINGS

- Two problems:
  - Strings can be of variable length
  - Strings can be long leading to low fanout (increased tree height)
- With variable-length search keys, different nodes can have different fanouts even if they are full
  - A node must be split if it is full (no space for a new entry) regardless of how many entries in the tree
- The fanout degree can be increased for strings with prefix compression
  - No need to store the entire search key value at non-leaf nodes
  - Store a prefix of each search key value that is sufficient to distinguish between the key values in subtrees it separates
  - Example: Store "Silb" instead of "Silberschatz" if the closest values in the two subtrees that it separates are "Silas" and "Silver"

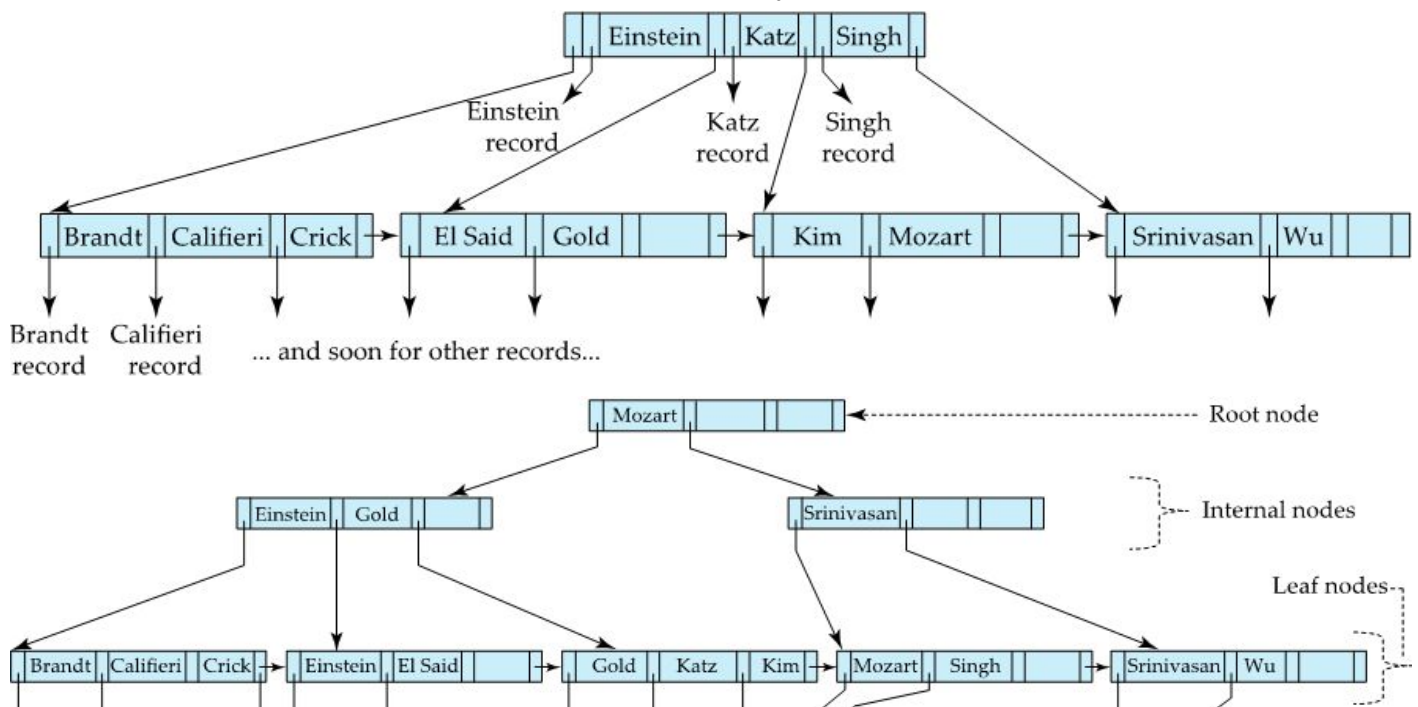
## BULK LOADING IN B+-TREE

- Insertion of one record in B+-tree is  $O(\log_{n/2}(N))$
- Suppose the relation to be indexed is significantly larger than main memory, so is the index. Also assume that this is a non-clustering index (records are not sequential w.r.t. search key). Thus, no particular order of search-key values scanned
  - For example, instructor is stored physically on disk ordered by ID, and we are building a B+-tree on dept\_name
- As we scan the relation and add entries to the B+-tree, it is probable that each leaf node will not be in the database buffer
- Each time an entry is added to the leaf, a disk seek will be required to fetch the block, and another one to write it back
- Efficient way of bulk loading of the index:
  - Create a temporary file containing index entries for the relation

- Sort the file on the search key
- Scan the sorted file and insert the entries into the index
- Why is inserting in sorted order beneficial?
  - All entries that go to a particular leaf node will appear consecutively and the leaf needs to be written out only once
  - Nodes will never have to be read from disk during bulk load if the B+-tree was empty
- If the B+-tree is initially empty, it can be constructed faster by building it bottom-up starting from the leaf level
- Entries are sorted
- Break up the sorted entries into blocks
  - This forms the leaf level of the B+-tree
- The minimum value of in each block (leaf node) along with the pointer to the block is used to create entries in the next level
- Each further level is similarly constructed using the minimum values associated with each node from the lower-level
- Most database systems implement efficient techniques using sorting of entries and bottom-up construction
- Normal insertion when tuples are added later
- Some systems drop the index and reconstruct it if there is a very large number of tuples to be added to an existing relation (non-empty index)

## B-TREE INDEX

- Similar to B+-tree, but it allows search-key values to appear only once to eliminate redundant storage of search-key values
- Search-key values in non-leaf nodes do not appear anywhere else in the B-tree
  - An additional pointer field for each search key in a non-leaf node must be included



- Advantages
  - Uses a smaller number of tree nodes (less space)
  - Possible to find search-key value before reaching leaf nodes
- Disadvantages
  - Only small fraction of all search-key values are found early
    - Roughly  $n$  times as many keys are stored in the leaf level of a B-tree as in the non-leaf levels
  - Non-leaf nodes are larger, so fan-out is reduced
    - B-tree typically have greater depth than corresponding B+-tree
  - Deletions are more complicated
    - Deleted entry can be in a non-leaf node
    - Proper replacement value must be selected
    - May require further action if the lower level node has too few entries

## INDEXING ON FLASH STORAGE

- Random I/O costs much less on flash
  - 20 to 100 microseconds for read/write in contrast to 5 to 10 milliseconds in magnetic disk
- The performance of write operations is more complicated with flash storage
  - Flash storage does not permit in-place updates at the physical level
  - Every update turns into a copy+write an entire flash-storage page
  - Old copy of the page will be erased later along with other pages in what is called erase blocks
  - Erasing a block of pages takes 2 to 5 milliseconds
- Flash pages are typically smaller than the magnetic disk page
- Optimum B+-tree node size for flash storage is smaller than that of magnetic disk
- Although smaller nodes lead to taller trees and more I/O operations to access data, random page reads are much faster
- Bulk loading is even more beneficial for flash storage
  - Reduces the number of page writes compared to tuple-at-a-time insertion

## INDEXING IN MAIN MEMORY

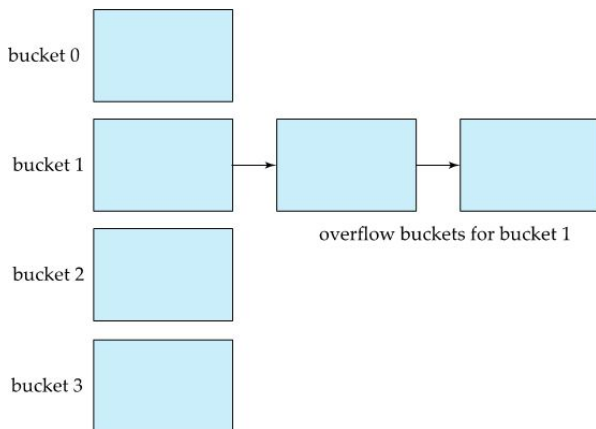
- B+-tree can be used to index in-memory data
- Memory is costlier than disk
- Reduce space requirements using techniques discussed earlier
- Traversal of in-memory pointers is acceptable unlike the case with disks
- Trees can be deep
- Maximize cache hits
  - Data is transferred between main memory and cache in unites of cache-lines (typically 64 bytes)
  - Treat the cache line the same way we treated disk block
- If data is present in cache, the CPU can complete a read in 1 or 2 nanoseconds
- If data is to be fetched from memory (cache miss), it takes 50 to 100 nanoseconds
- B+-tree with node size that fits in a cache line provides very good performance with in-memory data

## HASH INDICES

- A bucket is a unit of storage containing one or more entries
  - Typically the size of a disk block
  - We obtain the bucket of an entry from its search-key value using a hash function
- Hash function  $h$  is a function from the set of all search-key values  $K$  to the set of all bucket addresses  $B$
- Hash function is used to locate entries for access, insertion as well as deletion
- Entries with different search-key values may be mapped to the same bucket
  - Entire bucket has to be searched sequentially to locate an entry
- In hash index, buckets store entries with pointers to records
- In hash file-organization, buckets store records

## BUCKET OVERFLOWS

- Bucket overflow occurs because:
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to:
    - Multiple records have same search-key value
    - Chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated
- Overflow is handled through overflow buckets
- Overflow chaining: The overflow buckets of a given bucket are chained together in a linked list



## USE OF HASH INDICES

- Best used for equality queries
- To perform a lookup on a search-key value  $K_i$ , compute  $h(K_i)$ , then search the bucket with that address
- Not suitable for range queries

## MULTIPLE-KEY ACCESS

- Assume the instructor relation with two indices, one for dept\_name and one for salary
- Consider the following query:

```
select ID
from instructor
where dept_name = 'Finance' and salary = 80000;
```

- Three possible strategies:
  - Use the index on dept\_name to find all records pertaining to the finance department, then scan through records to find the instructor with salary = 80000
  - Use the index on salary to find all records pertaining to instructors with salary of \$80,000, then scan through records to find who of those is in the Finance department
- Previous two strategies use one index only
- Third strategy: Use the index on dept\_name to find pointers to all records pertaining to the Finance department, and do the same with the index on salary
  - The intersection of pointers pertain to the instructors with salary of \$80,000 in the Finance department
- The third strategy is the only one that takes advantage of the multiple indices

## INDICES ON MULTIPLE KEYS

- Alternatively, create an index on a composite search key (dept\_name, salary)
- For example, a B+-tree
- A query that has a range condition can also be efficiently processed:

```
select ID
from instructor
where dept_name = 'Finance' and salary < 80000;
```

- CREATING AND DROPPING INDICES
- Creating an index:

```
create index <index_name> on <relation_name> (<attribute_list>);
```

- Example:

```
create index dept_index on instructor (dept_name);
```

- Dropping an index:

```
drop index <index_name>;
```

- Query processor automatically uses the index
- Most database systems automatically create an index on the primary key
  - Whenever a tuple is inserted, it is efficient to check if a tuple with the same value of primary key exists
  - Without this index, the database would have to scan the whole table to check for violation of uniqueness



# Questions

Question Topics:

[https://drive.google.com/file/d/1yDV9oXvv-xDnBSZbbsCyldLX2ZqY\\_KDU/view?usp=sharing](https://drive.google.com/file/d/1yDV9oXvv-xDnBSZbbsCyldLX2ZqY_KDU/view?usp=sharing)

## Relational Algebra

- 

## SQL

- <https://www.w3resource.com/sql-exercises/>

Q 5 (BONUS):

(4 points)

As discussed in class, SQL does not support functional dependency constraints. But it supports materialized views. Assume that the DBMS maintains the materialized view immediately. Given a relation  $R(X, Y, Z)$ , how would you use materialized views to enforce the functional dependency  $Y \rightarrow Z$ ?

```
create materialized view V as
select distinct Y, Z
from R;
alter table V add constraint V_pk primary key (Y);
```

- 

- <https://www.interviewsansar.com/mcq-basic-sql-multiple-choice-questions-answers/>
- [https://www.tutorialspoint.com/sql/sql\\_questions\\_answers.htm](https://www.tutorialspoint.com/sql/sql_questions_answers.htm)

Q 7 (BONUS):

(3 points)

Consider the following schema:  
*branch*(*branch\_name*, *branch\_city*, *assets*)  
*customer*(*customer\_name*, *customer\_street*, *customer\_city*)  
*loan*(*loan\_number*, *branch\_name*, *amount*)  
*borrower*(*customer\_name*, *loan\_number*)  
*account*(*account\_number*, *branch\_name*, *balance*)  
*depositor*(*customer\_name*, *account\_number*)  
Write a trigger that enforces the following: On delete of an account, for each owner of the account, check if the owner has any remaining accounts, and if she does not, delete her from the depositor relation.

```
create trigger check_delete after delete on account
referencing old row as orow
for each row
delete from depositor
where depositor.customer_name not in
(select customer_name from depositor
where account_number <> orow.account_number)
end
```

- 

## Entity-Relations

- <http://math.hws.edu/bridgeman/courses/343/f18/practice/er.html>

## Normalization

Q 4:

(6 points)

How can you use functional dependencies to represent the constraint that a relationship between two entity sets  $X$  and  $Y$  is one-to-one? Just give the functional dependencies that represent the constraint. Repeat for a many-to-one relationship from  $X$  to  $Y$ .

One-to-one:  $PK(X) \rightarrow PK(Y)$  and  $PK(Y) \rightarrow PK(X)$

Many-to-one:  $PK(X) \rightarrow PK(Y)$

## Storage Organization Raid Questions

Q 1:

(4 points)

RAID systems can support replacing failed disks without the system going offline. Which of the RAID levels better support this operation with the least amount of interference between the rebuild and ongoing disk accesses? Explain your answer.

RAID level 1. This is because rebuilding the failed disk involves copying from only the mirror (shadow) of the failed disk. In other RAID levels discussed in class, this will require reading from all other disks.

### Other storage questions

Q 1:

(3 points)

Does the remapping of bad sectors by disk controllers affect data-retrieval rates? If yes, how? If no, why?

Remapping of bad sectors by disk controllers does reduce data-retrieval rates because of the loss of sequentiality among the sectors. But that is better than the loss of data in case of no remapping.

Q 2:

(12 points)

## Database structures

## Assignment #1

Instructor: Ahmed El-Roby

Name: , ID:

**Instructions:** Read all the instructions below carefully before you start working on the assignment, and before you make a submission.

- The accepted formats for your submission are: pdf, docx, txt. More details below.
- You can either write your solutions in the tex file (then build to pdf) or by writing your solution by hand or using your preferred editor (then convert to pdf or docx). However, you are encouraged to write your solutions in the tex file (5% bonus). If you decide not to write your answer in tex, it is your responsibility to make sure you write your name and ID on the submission file.
- If you use the tex file, make sure you edit line 28 to add your name and ID. Only write your solution and do not change anything else in the tex file. If you do, you will be penalized.
- All questions in this assignment use the university schema discussed in class (on culearn), unless otherwise stated.
- For SQL questions, upload a text file with your queries in the format shown in the file “template.txt” uploaded on culearn. An example submission is in the file “sample.txt”. You will be penalized if the format is incorrect or there is no text file submission.

**Q 1:**

(6 points)

Answer the following questions using the university schema discussed in class:

(a) The primary key for the *advisor* relation is *s\_id*. Suppose a student can have more than one supervisors. Would *s\_id* still be a primary key in *advisor*? If yes, why? If not, what would be a suitable primary key? (2 marks)

No, *s\_id* would not be a primary key, since there may be two (or more) tuples for a single student, corresponding to two (or more) advisors. The primary key should then consist of the two attributes *s\_id*, *i\_id*.

(b) The primary key for *prereq* is both attributes *course\_id* and *prereq\_id*. Why wouldn't only *course\_id* work as primary key? (2 marks)

Because a course could have multiple prerequisites. In this case, we would expect multiple tuples with the same *course\_id* and different *prereq\_id*. Since primary key is unique, this will not be possible if only *course\_id* is the primary key.

(c) Given the existing schema of *teaches*, two or more instructors can teach the same section. How can the primary key be changed to restrict a section to one instructor only? (2 marks)

The primary key would exclude *ID*. This means that there will be at most one tuple for each section. Hence, one instructor.

**Q 2:**

(12 points)

Consider the following bank database schema:

*branch*(*branch\_name*, *branch\_city*, *assets*)  
*customer*(*ID*, *customer\_name*, *customer\_street*, *customer\_city*)  
*loan*(*loan\_number*, *branch\_name*, *amount*)  
*borrower*(*ID*, *loan\_number*)  
*account*(*account\_number*, *branch\_name*, *balance*)  
*depositor*(*ID*, *account\_number*)

Write an expression in relational algebra to find the following:

(a) Find the cities that host branches that have a loan that is greater than \$50000. (3 marks)

$\Pi_{branch\_city} (branch \bowtie_{branch.branch\_name=loan.branch\_name} \sigma_{amount>50000} (loan))$

(b) Find the ID of each depositor who has an account with a balance greater than \$50000 at the “Nepean” branch. (3 marks)

$\Pi_{ID}(\sigma_{balance>10000 \wedge branch\_name=“Nepean”} (depositor \bowtie_{depositor.account\_number=account.account\_number} account))$

(c) Find the names of customers who have at least one loan amount that is greater than at least one account balance. (6 marks)

$\Pi_{customer\_name} (customer \bowtie_{customer.ID=depositor.ID} (\sigma_{loan.amount>account.balance} ((depositor \bowtie_{depositor.account\_number=account.account\_number} account) \bowtie_{depositor.ID=borrower.ID} (borrower \bowtie_{borrower.loan\_number=loan.loan\_number} loan))))$ .

**Q 3:**

(33 points)

Using the university database schema discussed in class, write the SQL statements that do:

(a) Create a new course (“Aces of Databases”) with ID (“COMP5118”) in the Computer Science department (“Comp. Sci.”) with 0 credit hours. (3 marks)

```
insert into course
values ('COMP5118', 'Aces of Databases', 'Comp. Sci.', 0);
```

(b) Create a section 'A' for this course in the Winter of 2020 with no known location or time, yet. (4 marks)

```
insert into section
values ('COMP5118', 'A', 'Winter', 2020, null, null, null)
```

(c) Enroll all students in the department into this course. (5 marks)

```
insert into takes(
select ID , 'COMP5118', 'A', 'Winter', 2020, null
from student
where dept name = 'Comp. Sci.');
```

(d) One student with ID 12345 cannot take this course because of violating the prerequisite requirements (didn't pass COMP3005). Unregister this student from the new section. (3 marks)

```
delete from takes
where course\_id= 'COMP5118' and sec\_id = 'A' and
year = 2020 and semester = 'Winter' and ID = 12345;
```

(e) For each student who took a course at least twice, show the course ID and the student ID. (5 marks)

```
select distinct course\_id, ID
from takes
group by ID , course\_id
having count(*) >= 2;
```

(f) Find the ID and name of instructors who never gave a grade 'A' in the courses they taught (note that instructors who never taught a course satisfy this condition). (5 marks)

```
select ID , name
from instructor
except (
select distinct instructor.ID , instructor.name
from instructor, teaches, takes
where instructor.ID = teaches.ID
and teaches.course_id = takes.course_id
and teaches.year = takes.year
and teaches.semester= takes.semester
and teaches.sec_id= takes.sec_id
and takes.grade = 'A';)
```

(g) Rewrite the previous query so that you make sure that the instructor taught at least one course. (5 marks)

```
select distinct(instructor.ID), instructor.name
from instructor, teaches, takes
where instructor.ID =teaches.ID
and teaches.course_id=takes.course_id
and teaches.year = takes.year
and teaches.semester = takes.semester
and teaches.sec_id = takes.sec_id
and takes.grade is not null
except
select distinct instructor.ID , instructor.name
from instructor, teaches, takes
where instructor.ID = teaches.ID
and teaches.course_id = takes.course_id
and teaches.year = takes.year
and teaches.semester = takes.semester
and teaches.sec_id = takes.sec_id
and takes.grade= 'A';
```

without except

```
select distinct instructor.ID , instructor.name
from instructor, teaches, takes
where instructor.ID = teaches.ID
and teaches.course_id = takes.course_id
and teaches.year = takes.year
and teaches.semester = takes.semester
and teaches.sec_id = takes.sec_id
and takes.grade is not null
and instructor.ID not in (
select instructor.ID
from instructor, teaches, takes
where instructor.ID = teaches.ID
and teaches.course_id = takes.course_id
and teaches.year = takes.year
and teaches.semester = takes.semester
and teaches.sec_id = takes.sec_id
and takes.grade = 'A'
);
```

(h) Find the lowest, across all departments, of the per-department maximum salary. (3 marks)

```
select min(maxsalary)
```

```
from (select dept_name, max(salary) as maxsalary
from instructor
group by dept_name);
```

**Q 4**

(5 points)

Consider the following car insurance schema:

*person*(driver\_id, name, address)

*car*(licence, model, year)

*accident*(report\_number, date, location)

*owns*(driver\_id, licence\_plate)

*participated*(report\_number, licence\_plate, driver\_id, damage\_amount)

Write SQL queries to:

(a) Find the number of accidents involving a car belonging to a person named “Ahmed El-Roby”. (3 marks)

```
select count (distinct report_number)
from participated, owns, person
where owns.driver_id = person.driver_id
and person.name = 'Ahmed El-Roby'
and owns.licence_plate = participated.licence_plate;
```

(b) Update the damage amount for the car with licence plate “DB007” in the accident with report number “AR2020” to \$3000. (2 marks)

```
update participated
set damage_amount = 3000
where report_number = 'DB007' and
licence_plate = 'AR2020');
```

## Assignment #2

Instructor: Ahmed El-Roby

Name: , ID:

**Instructions:** Read all the instructions below carefully before you start working on the assignment, and before you make a submission.

- The accepted formats for your submission are: pdf, docx, txt, and java. More details below.
- You can either write your solutions in the tex file (then build to pdf) or by writing your solution by hand or using your preferred editor (then convert to pdf or docx). However, you are encouraged to write your solutions in the tex file (5% bonus). If you decide not to write your answer in tex, it is your responsibility to make sure you write your name and ID on the submission file.
- If you use the tex file, make sure you edit line 28 to add your name and ID. Only write your solution and do not change anything else in the tex file. If you do, you will be penalized.
- All questions in this assignment use the university schema discussed in class (on culearn), unless otherwise stated.
- For SQL questions, upload a text file with your queries in the format shown in the file “template.txt” uploaded on culearn. An example submission is in the file “sample.txt”. You will be penalized if the format is incorrect or there is no text file submission.
- For programming questions, upload your .java file.
- Late submissions are allowed for 24 hours after the deadline above with a penalty of 10% of the total grade of the assignment. Submissions after more than 24 are not allowed.

**Q 1:**

(3 points)

Consider the following DDL statements:

```
create table takes
  (ID          varchar(5),
   course_id   varchar(8),
   sec_id      varchar(8),
   semester    varchar(6),
   year        numeric(4,0),
   grade       varchar(2),
   primary key (ID, course_id, sec_id, semester, year),
   foreign key (course_id, sec_id, semester, year) references section
     on delete cascade,
   foreign key (ID) references student
     on delete cascade
);

create table section
  (course_id   varchar(8),
   sec_id      varchar(8),
   semester    varchar(6)
     check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
   year        numeric(4,0)
     check (year > 1701 and year < 2100),
   building    varchar(15),
   room_number varchar(7),
```

```

time_slot_id      varchar(4),
primary key (course_id, sec_id, semester, year),
foreign key (course_id) references course
    on delete cascade,
foreign key (building, room_number) references classroom
    on delete set null
);

```

Now, consider the following SQL query:

```

select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2017
group by course_id, semester, year, sec_id
having count (ID) >= 2;

```

Will appending **natural join section** in the **from** clause change the returned result? Explain why?

Adding a natural join with section would remove from the result each tuple in *takes* whose values for (*course\_id*, *semester*, *year*, *sec\_id*) do not appear in *section*. However, since *takes* has the constraint:

**foreign key (course\_id, semester, year, sec\_id) references section**

there cannot be a tuple in *takes* whose values for (*course\_id*, *semester*, *year*, *sec\_id*) do not appear in *section*.

**Q 2:**

(2 points)

Write an SQL query using the university schema to find the names of each instructor who has never taught a course at the university. Do this using no subqueries and no set operations.

```

select name
from instructor left outer join teaches using (ID)
where course_id is null;

```

This query can also use **natural left outer join**, or using **on**.

**Q 3:**

(2 points)

Rewrite the following query to replace the natural join with an inner join with **using** condition:

```

select *
from section natural join classroom;

```

```

select *
from section join classroom using (building, room_number);

```

**Q 4:**

(5 points)

Using the university schema, define a view *tot\_credits* (*year*, *num\_credits*), giving the total number of credits taken in each year. Then, explain why insertions would not be possible into this view.

```

create view tot_credits(year, num_credits) as
select year, sum(credits)
from takes natural join course
group by year;

```

In this query, that will be due to the aggregate function (*sum*). Insertions will fail because there are many ways to insert values into the *credits* attribute for each tuple given a sum of all these values.

**Q 5:**

(10 points)

Write a Java program that finds all prerequisites for a given course using JDBC. The program should:



- Takes a course id value as input using keyboard.
- Finds the prerequisites of this course through a SQL query.
- For each course returned, repeats the previous step until no new prerequisites can be found.
- Prints the results.

Don't forget to handle the case for cyclic prerequisites. For example, if course A is prerequisite to course B, course B is prerequisite to course C, and course C is prerequisite to course A, do not infinite loop.

```
import java.sql.*;
import java.util.Scanner;
import java.util.Arrays;
public class AllCoursePrereqs {
    public static void main(String[] args) {
        try {
            Connection con = DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/univ_db","userid","passwd");
            Statement s = con.createStatement();
        ){
            String q;
            String c;
            ResultSet result;
            int maxCourse = 0;
            q = "select count(*) as C from course";
            result = s.executeQuery(q);
            if (!result.next())
                System.out.println("Unexpected empty result.");
            else
                maxCourse = Integer.parseInt(result.getString("C"));
            int numCourse = 0, oldNumCourse = -1;
            String[] prereqs = new String [maxCourse];
            Scanner krb = new Scanner(System.in);
            System.out.print("Input a course id (number): ");
            String course = krb.next();
            String courseString = "" + '\'' + course + '\'';
            while (numCourse != oldNumCourse) {
                for (int i = oldNumCourse + 1; i < numCourse; i++) {
                    courseString += ", " + '\'' + prereqs[i] + '\'' ;
                }
                oldNumCourse = numCourse;
                q = "select prereq_id from prereq where course_id in ("
                    + courseString + ")";
                result = s.executeQuery(q);
                while (result.next()) {
                    c = result.getString("prereq_id");
                    boolean found = false;
                    for (int i = 0; i < numCourse; i++)
                        found |= prereqs[i].equals(c);
                    if (!found) prereqs[numCourse++] = c;
                }
                courseString = "" + '\'' + prereqs[oldNumCourse] + '\'';
            }
            Arrays.sort(prereqs, 0, numCourse);
            System.out.print("The courses that must be taken prior to "
                + course + " are: ");
            for (int i = 0; i < numCourse; i++)
                System.out.print ((i==0?" ":"") + prereqs[i]);
            System.out.println();
        }
    }
}
```

```

    } catch(Exception e){e.printStackTrace();
  }
}

```

**Q 6:**

(5 points)

Consider the following schema:

*employee*(*emp\_name*, *street*, *city*)

*works*(*emp\_name*, *company\_name*, *salary*)

Write a function *avg\_sal* that takes a company name as input and finds the average salary of employees in the company. Then, write a SQL query that uses this function to find companies whose employees earn (on average) higher salary than the company “Losers Inc.”.

```

create function avg_sal(cname varchar(15))
returns numeric(10, 2)
declare result numeric(10,2);
    select avg(salary) into result
    from works
    where works.company_name = cname
return result;
end

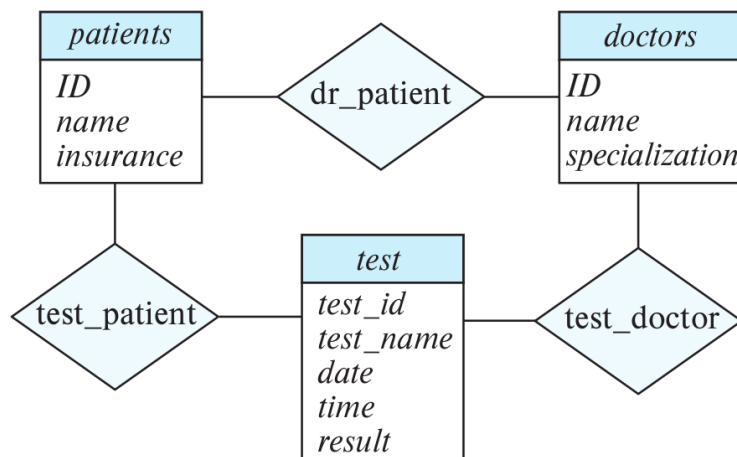
select company_name
from works
where avg_sal(company_name) > avg_sal('Losers Inc.')

```

**Q 7:**

(10 points)

Design a database for a hospital with a set of patients and a set of medical doctors. The database should capture each encounter between a doctor and a patient on each visit. Associate with each patient a log of the various tests and examinations conducted. Those tests can be ordered by a doctor, but this is not mandatory. The database should keep track of the dates of the tests as well as their results. Draw an ER-diagram for your design. Underline the primary key in the entity sets.



There are two missing attributes for the *dr\_patient* relationship set. Those are *date* and *time*.

**Q 8:**

(6 points)

Reduce your ER-diagram to relations. Underline primary keys.

*patients*(*ID*, *name*, *insurance*)

*doctors*(*ID*, *name*, *specialization*)

*test*(*test\_id*, *test\_name*, *date*, *time*, *result*)

*dr\_patient(d\_id, p\_id, date, time)*

*test\_doctor(d\_id, t\_id)*

*test\_id(p\_id, t\_id)*

**Q 9:**

(5 points)

If any test must be ordered by only one doctor, what needs to be changed in your answer to Q7? Either draw a new ER-diagram, or describe the changes. How will the reduced relations change?

The participation of *test* in *test\_doctor* will be total.

Relation *test\_doctor* is no longer needed and *test\_id*(p\_id, t\_id, *d\_id*).

## Midterm Exam

Instructor: Ahmed El-Roby

Name: \_\_\_\_\_, ID: \_\_\_\_\_

**Instructions:**

- Please write your name on each page in this exam booklet. That is, in the above box on this page, and on the top left of each following page.
- This is a closed-book exam. No course material is allowed during the exam.

**Q 1:**

(6 points)

Write the following queries in **relational algebra**, using the university schema in Figure 2:

(a) Find the names of advisors who supervise students from the “Comp. Sci.” department.

$$\Pi_{name}(instructor \bowtie_{instructor.ID=advisor.i\_id} (advisor \bowtie_{student.ID=advisor.s\_id} (\sigma_{dept\_name="Comp.Sci."}(student))))$$

(b) Find the course ID and title of courses that were never taught by instructors from the same department as that of the course.

$$\Pi_{course\_id,title}(course) -$$

$$\Pi_{course\_id,title}$$

$$(course \bowtie_{teaches.course\_id=course.course\_id \wedge instructor.dept\_name=course.dept\_name} (instructor \bowtie_{instructor.ID=teaches.ID} teaches))$$

**Q 2:**

(12 points)

Using the **university schema** in Figure 2, write **SQL** queries to display the following:

(a) Id's of instructors who advise more than 5 students.

```
select i_id, count(s_id)
from advisor
group by i_id
having count(s_id) > 5
```

(b) Titles of courses that have been taught only in the fall (never been taught in any other term).

```
select course_id, title
from course natural join (select course_id
from section
except
(select distinct course_id
from section
where semester <> 'Fall')) as non_fall_courses
```

(c) Using the **university schema** in Figure 2, write a query to display the list of all departments, along with their budgets, that have more than 12 instructors.

```
select dept_name, budget
from department natural join
(select dept_name, count(*)
from instructor
group by dept_name
having count(*) > 12) as dept_inst_count
```

**Q 3:**

(12 points)

Using only binary relationships, modify the ER-Diagram in Figure 3 to model the need for associating student exams with sections of courses. Note that there may be multiple exams in any section (e.g., midterm and final). The new model should keep track of marks of exams for all students writing these exams and the exam type. It is possible that a student does not write an exam (absent). **Draw only the modified part of the ER diagram. Reduce any entity set or relationship set you modified or created to relations.**

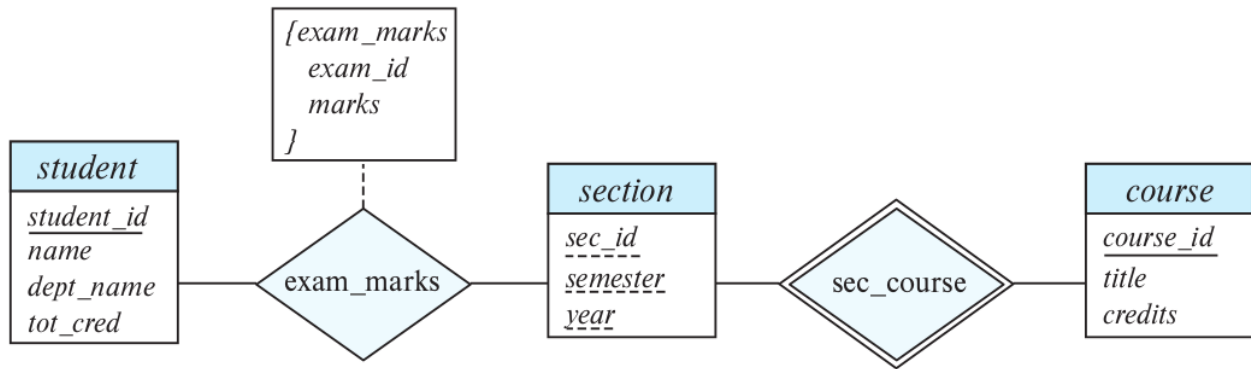


Figure 1: Solution 1 for ER question.

*exam\_marks(student\_id, course\_id, sec\_id, semester, year, exam\_id, marks)*

**Q 4:**

(6 points)

Consider the following instance of relation  $R(A, B, C)$ :

$(a_1, b_1, c_1)$

$(a_1, b_2, c_1)$

$(a_2, b_1, c_1)$

$(a_2, b_3, c_1)$

List all **nontrivial** functional dependencies that hold on this relation instance.

$A \rightarrow C$

$B \rightarrow C$

$AB \rightarrow C$

**Q 5:**

(4 points)

Suppose we decompose the schema  $R = (A, B, C, D, E)$  into  $R_1 = (A, B, C)$  and  $R_2 = (A, D, E)$ . Assume the following set of functional dependencies for  $R$ :  $F = \{A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A\}$ . Is this decomposition lossy or lossless? Why?

This is a lossless decomposition because either  $R_1 \cap R_2 \rightarrow R_1$  and  $R_1 \cap R_2 \rightarrow R_2$  hold.

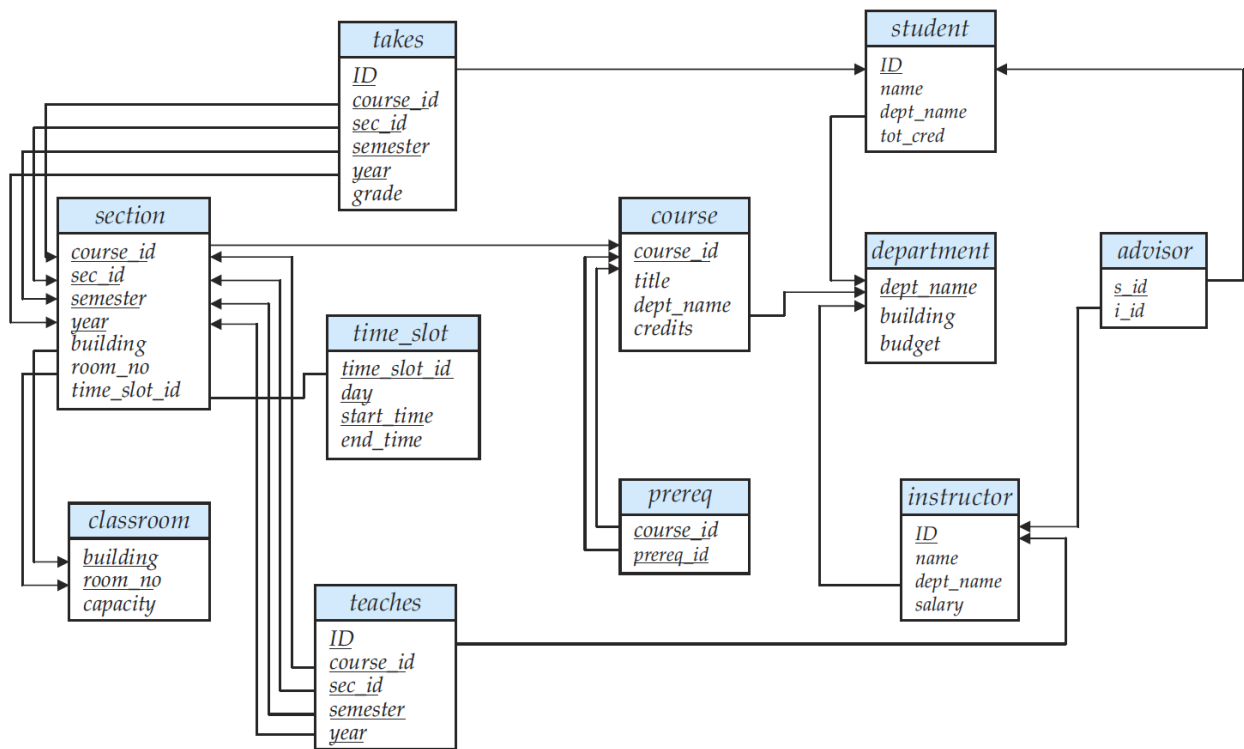


Figure 2: The university schema.



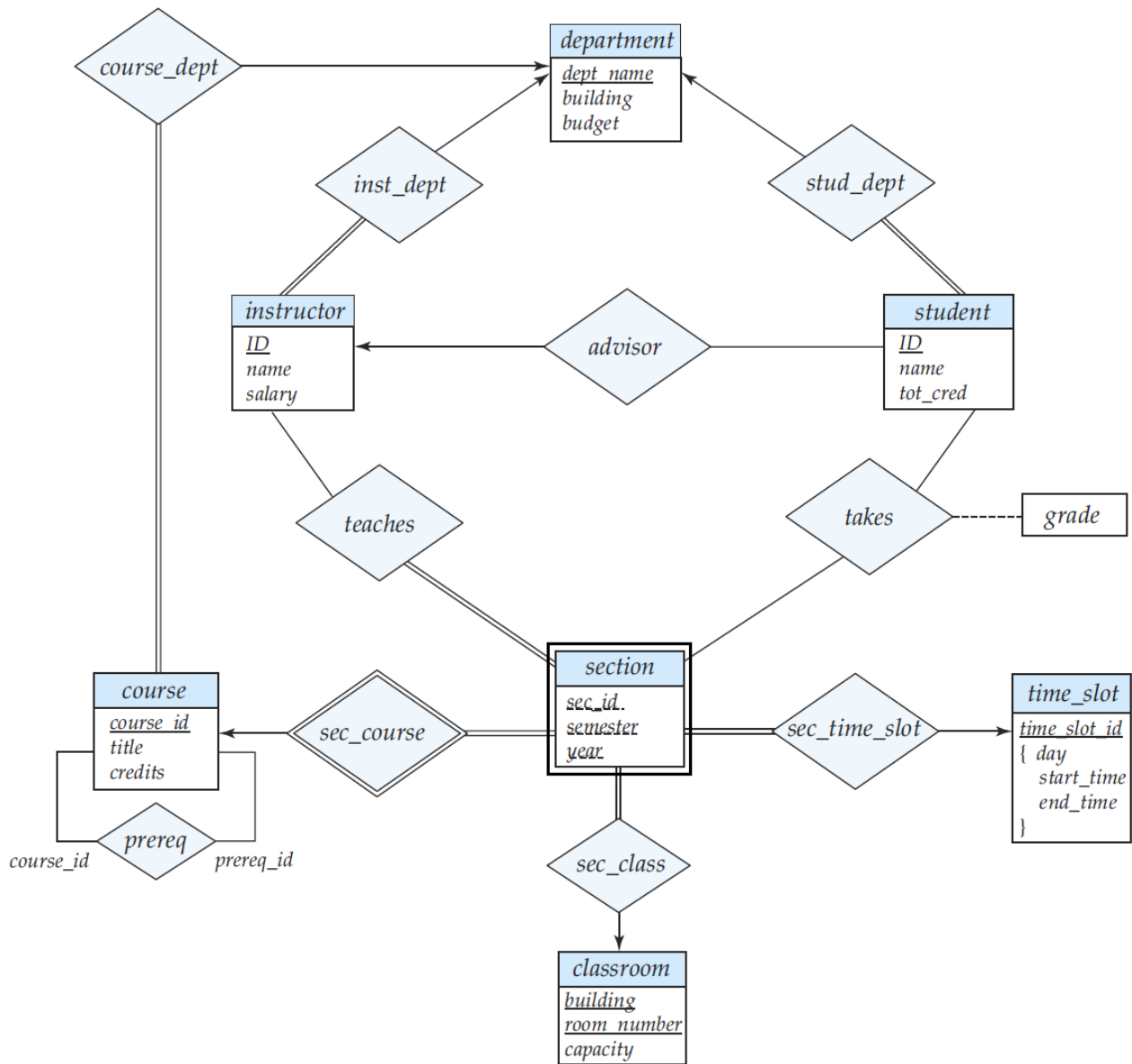


Figure 3: ER-Diagram for the university schema.

## Assignment #3

Instructor: Ahmed El-Roby

Name: , ID:

**Instructions:** Read all the instructions below carefully before you start working on the assignment, and before you make a submission.

- The accepted formats for your submission are: pdf and docx. More details below.
- If you use the tex file, make sure you edit line 28 to add your name and ID. Only write your solution and do not change anything else in the tex file. If you do, you will be penalized.
- Late submissions are allowed for 24 hours after the deadline above with a penalty of 10% of the total grade of the assignment. Submissions after more than 24 are not allowed.

**Q 1:**

(4 points)

Consider the following proposed rule: If  $A \rightarrow B$  and  $C \rightarrow B$ , then  $A \rightarrow C$ . Prove that this rule is not sound.

Proof by counter example:

$A$	$B$	$C$
$a_1$	$b_1$	$c_1$
$a_1$	$b_1$	$c_2$

**Q 2:**

(8 points)

Consider the following relation  $R = \{A, B, C, D, E\}$  and the following set of functional dependencies

$F = \{$   
 $A \rightarrow BC$   
 $CD \rightarrow E$   
 $B \rightarrow D$   
 $E \rightarrow A\}$

Compute  $B^+$ . Is  $R$  in BCNF? If not, give a lossless decomposition of  $R$  into BCNF. Show your work for all previous questions.

$B^+ = \{B, D\}$

The candidate keys are  $A, BC, CD$ , and  $E$ . See below:

$A \rightarrow BC$ , so  $A \rightarrow B$  and  $A \rightarrow C$ . Since  $A \rightarrow B$  and  $B \rightarrow D$ , then  $A \rightarrow D$

Since  $A \rightarrow CD$  and  $CD \rightarrow E$ , then  $A \rightarrow E$

Since  $A \rightarrow A$ , and from above  $A \rightarrow BCDE$ , then  $A \rightarrow ABCDE$

Since  $E \rightarrow A$ , then  $E \rightarrow ABCDE$

Since  $CD \rightarrow E$ , then  $CD \rightarrow ABCDE$

Since  $B \rightarrow D$  and  $BC \rightarrow CD$ , then  $BC \rightarrow ABCDE$

Now, from  $F$ ,  $B \rightarrow D$  is non-trivial and  $B$  is not superkey. Using the algorithm discussed in class, we derive the relations  $R_1(A, B, C, E)$  and  $R_2(B, D)$

**Q 3:**

(4 points)

Give a lossless, dependency-preserving decomposition into 3NF of schema  $R$  in Q2.

$F$  already forms a canonical cover. Using the algorithm discussed in class,  $R$  is decomposed into  $R_1(A, B, C)$ ,  $R_2(C, D, E)$ ,  $R_3(B, D)$ , and  $R_4(E, A)$ . No redundant relations exist.

**Q 4:**

(4 points)

Assume the following decomposition of  $R$  in Q2:  $R_1(A, B, C)$  and  $R_2(C, D, E)$ . Is this decomposition lossy or lossless? Why?

The decomposition is lossy because either  $R_1 \cap R_2 \rightarrow R_1$  or  $R_1 \cap R_2 \rightarrow R_2$  do not hold (look at answer to Q1).

**Q 5:**

(22 points)

Consider the following relation  $R(A, B, C, D, E, G)$  and the set of functional dependencies

$F = \{$   
 $A \rightarrow BCD$   
 $BC \rightarrow DE$   
 $B \rightarrow D$   
 $D \rightarrow A\}$

Note: Show the steps for each answer.

(a) Compute  $B^+$ . (4 points)

$B \rightarrow BD$  (third dependency)

$BD \rightarrow ABD$  (fourth dependency)

$ABD \rightarrow ABCD$  (first dependency)

$ABCD \rightarrow ABCDE$  (second dependency)

$B^+ = ABCDE$

(b) Prove (using Armstrong's axioms) that  $AG$  is superkey. (4 points)

$A \rightarrow BCD$  (Given)

$A \rightarrow ABCD$  (Augmentation with  $A$ )

$BC \rightarrow DE$  (Given)

$ABCD \rightarrow ABCDE$  (Augmentation with  $ABCD$ )

$A \rightarrow ABCDE$  (Transitivity)

$AG \rightarrow ABCDEG$  (Augmentation with  $G$ )

(c) Compute  $F_c$ . (6 points)

$D$  is extraneous in the first and second functional dependencies because of the third functional dependency. So, the new set of functional dependencies become:

$A \rightarrow BC$

$BC \rightarrow E$

$B \rightarrow D$

$D \rightarrow A$

We also note that  $C$  is extraneous in the second functional dependency because  $B \rightarrow E$  can be inferred using  $F$  (look at  $B^+$ ). So, the final canonical cover becomes:

$F_c =$

$A \rightarrow BC$

$B \rightarrow E$

$B \rightarrow D$

$D \rightarrow A$

Second and third functional dependencies can be also combined to be  $B \rightarrow DE$ . But this is optional.

(d) Give a 3NF decomposition of the given schema based on a canonical cover. (4 points)

Depending whether second and third dependencies were unioned. We could have two answers based on the algorithm discussed in class:  $R_1(A, B, C)$

$R_2(B, E)$

$R_3(B, D)$

$R_4(D, A)$

OR

$R_1(A, B, C)$

$R_2(B, D, E)$

$R_3(D, A)$

Neither of the above solutions has a candidate key because  $G$  is not dependent on any attribute. So, we create

another relation  $R_{4OR5}(A, G)$ .

(e) Give a BCNF decomposition of the given schema based on  $F$ . Use the first functional dependency as the violator of the BCNF condition. (4 points)

Starting with  $R$ , we see that it is not in BCNF because of the first functional dependency ( $A$  is not superkey). So, we decompose  $R$  based on the algorithm discussed in class into:

$R_1(A, B, C, D)$  and  $R_2(A, E, G)$ .

We then notice that  $A \rightarrow E$  is in  $F^+$  and causes  $R_2$  to violate BCNF. So, we decompose  $R_2$  further into:  $R_3(A, G)$  and  $R_4(A, E)$ . We now see that  $R_1$ ,  $R_3$ , and  $R_4$  are all in BCNF.

**Q 6:**

(6 points)

Given the following set of functional dependencies:

$A \rightarrow BC$

$B \rightarrow AC$

$C \rightarrow AB$

Show that it is possible to find more than one unique canonical cover for this set.

Consider the first functional dependency. We can verify that  $C$  is extraneous in  $A \rightarrow BC$  and delete it. Subsequently, we can similarly check that  $A$  is extraneous in  $B \rightarrow AC$  and delete it, and that  $B$  is extraneous in  $C \rightarrow AB$  and delete it, resulting in a canonical cover  $A \rightarrow B$ ,  $B \rightarrow C$ , and  $C \rightarrow A$ .

However, we can also verify that  $B$  is extraneous in  $A \rightarrow BC$  and delete it. Subsequently, we can similarly check that  $C$  is extraneous in  $B \rightarrow AC$  and delete it, and that  $A$  is extraneous in  $C \rightarrow AB$  and delete it, resulting in a canonical cover  $A \rightarrow C$ ,  $B \rightarrow A$ , and  $C \rightarrow B$ .

**Q 7:**

(7 points)

Consider the schema  $R = (A, B, C, D, E, G)$  and the set  $F$  of functional dependencies:

$A \rightarrow BC$

$BD \rightarrow E$

$CD \rightarrow AB$

Use the BCNF decomposition algorithm to find a BCNF decomposition of  $R$ . Start with  $A \rightarrow BC$ . Explain your steps. Is this decomposition lossy or lossless? Is it dependency-preserving?

First use  $A \rightarrow BC$  to decompose  $R$  into  $R_1 = (A, B, C)$  and  $R_2 = (A, D, E, G)$ . Then use the inferred  $AD \rightarrow E$  to decompose  $R_2$  into  $R_3 = (A, D, E)$  and  $R_4 = (A, D, G)$ . The resulting decomposition is  $(A, B, C)$ ,  $(A, D, E)$ ,  $(A, D, G)$ .

It is lossless since when the algorithm decomposes a schema using  $\alpha \rightarrow \beta$ , the attributes in  $\alpha$  appear in both resulting schemas ( $\alpha \cap \beta = \phi$ ) and can be used to join two schemas without introducing more tuples, since  $\alpha \rightarrow \alpha\beta$ .

It is not dependency preserving. It preserves  $A \rightarrow BC$ ,  $AD \rightarrow E$ , but not  $BD \rightarrow E$ .

## Assignment #4

Instructor: Ahmed El-Roby

Name: , ID:

**Instructions:** Read all the instructions below carefully before you start working on the assignment, and before you make a submission.

- The accepted formats for your submission are: pdf and docx. More details below.
- If you use the tex file, make sure you edit line 28 to add your name and ID. Only write your solution and do not change anything else in the tex file. If you do, you will be penalized.
- Late submissions are allowed for 24 hours after the deadline above with a penalty of 10% of the total grade of the assignment. Submissions after more than 24 are not allowed.

Q 1:

(3 points)

In variable-length record representation, the record starts with offset and length pairs of variable-size attributes, followed by fixed-size attributes, then the null bitmap, and finally the variable-size attributes. How can we improve this representation if our application is expected to store tables with large number of attributes, most of which are nulls?

We should be able to locate the null bitmap and the offset and length values of non-null attributes using the null bitmap. This can be done by storing the null bitmap at the beginning and then for non-null attributes, store the value (for fixed size attributes), or offset and length values (for variable sized attributes) in the same order as in the bitmap, followed by the values for non-null variable sized attributes. This representation is space efficient but needs extra work to retrieve the attributes.

Q 2:

(4 points)

Consider the following arrangement for four disks, where  $B_i$  is a data block, and  $P_i$  is the parity block for the 4 data blocks that precedes it. What problem will this arrangement cause?

Disk 1	Disk 2	Disk 3	Disk 4
$B_1$	$B_2$	$B_3$	$B_4$
$P_1$	$B_5$	$B_6$	$B_7$
$B_8$	$P_2$	$B_9$	$B_{10}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

The problem is that one block and its parity block will reside on the same disk. That is, the parity block  $P_i$  and  $B_{4i-3}$  are on the same disk. When this disk fails, reconstruction of  $B_{4i-3}$  won't be possible.

Q 3:

(5 points)

Consider the following file organization using free list.

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Show the structure of the file after each of the following operations (they follow each other):

(a) Delete record 9. (2 marks)

Record 9 is deleted.

Header points to record 1.

Record 1 points to record 4.

Record 4 points to record 6.

Record 6 points to record 9. Record 9's pointer is set to null.

(b) Insert (20000, Jamie, Physics, 100000). (3 marks)

New record is inserted in record 1.

Header points to record 4.

Record 4 points to record 6.

Record 6 points to record 9.

**Q 4:**

(12 points)

Construct a  $B^+$ -tree for the following set of key values: (2, 3, 5, 7, 11, 17, 19, 23, 29, 31). The tree is initially empty and values are added one value at a time in ascending order. Consider the following values of  $n$ :

(a)  $n = 4$ . (4 points)

(b)  $n = 6$ . (4 points)

(c)  $n = 8$ . (4 points)

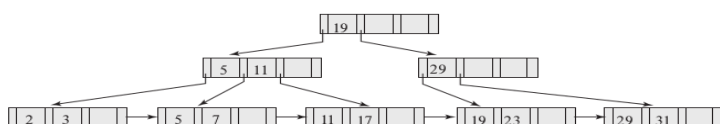


Figure 1: Q4.a ( $n = 4$ )

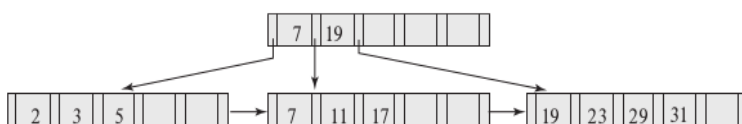


Figure 2: Q4.b ( $n = 6$ )

**Q 5:**

(8 points)

Consider the following  $B^+$ -tree with  $n = 4$ :

(a) Delete 23. (4 points)

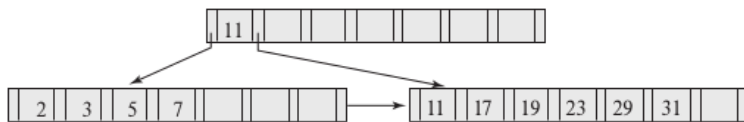
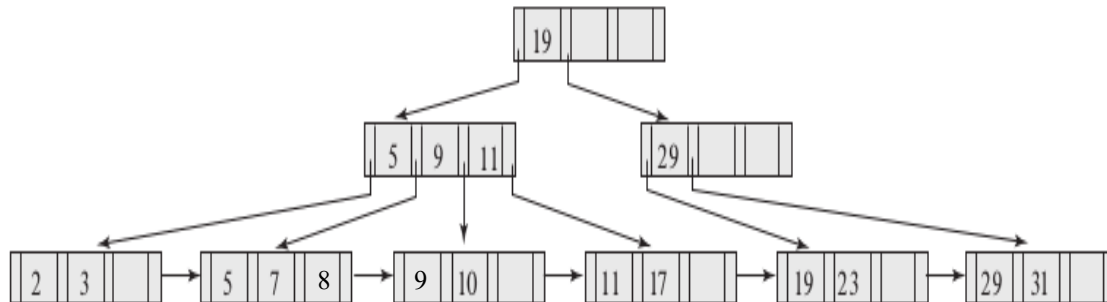


Figure 3: Q4.c ( $n = 8$ )



(b) Delete 19. (4 points)

**Q 6:**

(4 points)

Consider the following  $B^+$ -tree with  $n = 6$ : Insert 8 into this tree.

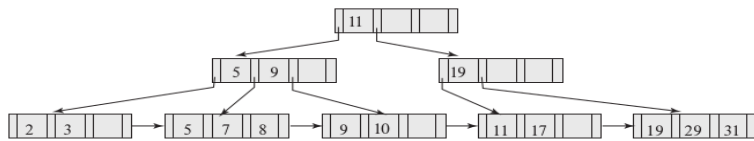


Figure 4: Q5.a (delete 23).

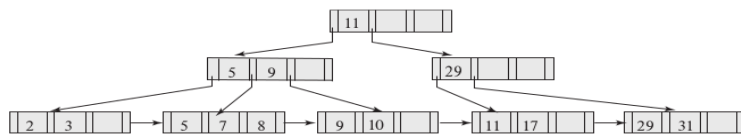


Figure 5: Q5.b (delete 19).

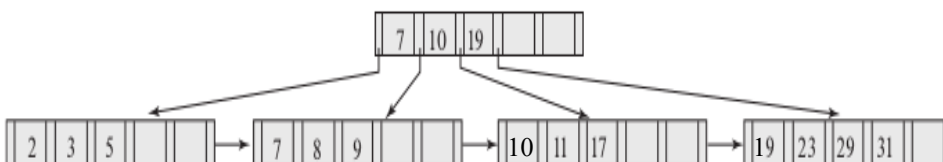
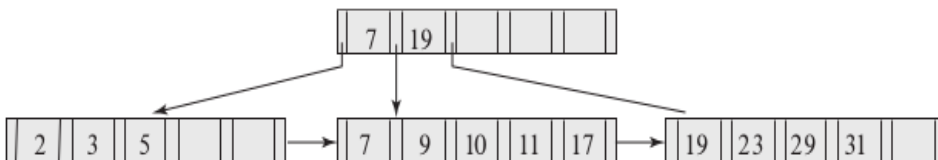


Figure 6: Q6 (Insert 8)



