Database Management Systems

By Anupratee Bharadwaj

Introduction to DBMS

Database Applications:

- Banking: transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Sales: customers, products, purchases
- Online retailers: order tracking, customized recommendations
- Manufacturing: production, inventory, orders, supply chain
- HR: employee records, salaries, tax deductions

Physical Data Management/ Book Keeping: Process of physical data or records management using physical ledgers and journals

Electronic Data Management (in order of evolution of data management):

- Punch cards/ data storage using tapes/magnetic tapes
- Spreadsheets/ Hard disks for direct access to data
- Relational DBMS
- Parallel and distributed databases
- Internet data management
- Unstructured data management became popular, NoSQL
- Data warehousing: integrates data and information collected from various sources into one comprehensive database

Spreadsheets vs Book Keeping:

- Durability – less prone to physical damage
- Scalability – easier to search and modify, easier to upgrade storage
- Security – password protected, user-based access
- Ease of Use – easier to search records, modify and maintain
- Consistency – less prone to mistakes, easier to check for them

DBMS vs File Systems/ Spreadsheets:

- No upper limit on no. of rows
- Faster to perform operations on huge amounts of data
- Ensure consistency of data
- Check violations of constraints (if a given value needs to be a string, make sure it is one)
- User based access/ permissions system
- Redundancies to protect from system crashes

File Handling via Python vs DBMS (IMPORTANT):

Parameter	File Handling via Python	DBMS
Scalability, amount of data	Very difficult to insert, modify and query records	In-built features to provide high scalability for a large number of records
Scalability, changes in data structure	Extremely difficult to change the structure of records as in the case of adding or removing attributes	Adding or removing attributes can be done seamlessly using simple SQL queries
Time of execution	In seconds	In milliseconds
Persistence, data survives after the process with which it was created has ended	Data processed using temporary data structures have to be manually updated to the file	Data persistence is ensured via automatic, system induced mechanisms
Robustness	Ensuring robustness of data has to be done manually	Backup, recovery and restore need minimum manual intervention
Security	Difficult to implement in Python (Security at OS level)	User-specific access at database level
Programmer's productivity	Most file access operations involve extensive coding to ensure persistence, robustness and security of data	Standard and simple built-in queries reduce the effort involved in coding thereby increasing a programmer's throughput
Arithmetic operations	Easy to do arithmetic computations	Limited set of arithmetic operations are available
Costs	Low costs for hardware, software and human resources	High costs of hardware, software and human resources

Levels of Abstraction (IMPORTANT):

- Physical Level – describes how a record is stored physically
- Logical Level – describes data stored in database and the relationships among the data fields, schema
- View Level: application programs hide details of data types, can also hide information for security purposes.

Schema & Instances:

- Schema: how the data is organised in a database, similar to type information of a variable
    1. Logical Schema – the overall logical structure of the database, define the data elements and their relationship
    2. Physical Schema – the overall physical structure of the database, how the actual database is built

- Physical Data Independence: the ability to modify the physical schema without changing the logical schema.

- Instance: the actual content of the database at a particular point, similar to the value of the variable

Data Definition Language:

- Defines the database schema
- Create, alter, drop objects in a database
- Grant, revoke privileges and roles
- A DDL compiler generates a set of table templates stored in a data dictionary (contains metadata, database schema, integrity constraints, authorizations)

Data Manipulation Language (Query Language):

- Language for accessing and manipulating the data organised by the appropriate data model
- Insert, delete
- Pure (relational algebra)
- Commercial (SQL)

Structured Query Language (SQL):

- Most widely used commercial language
- NOT a Turing Machine equivalent language (capable of performing complex functions and computations)

Database Design: logical and physical design

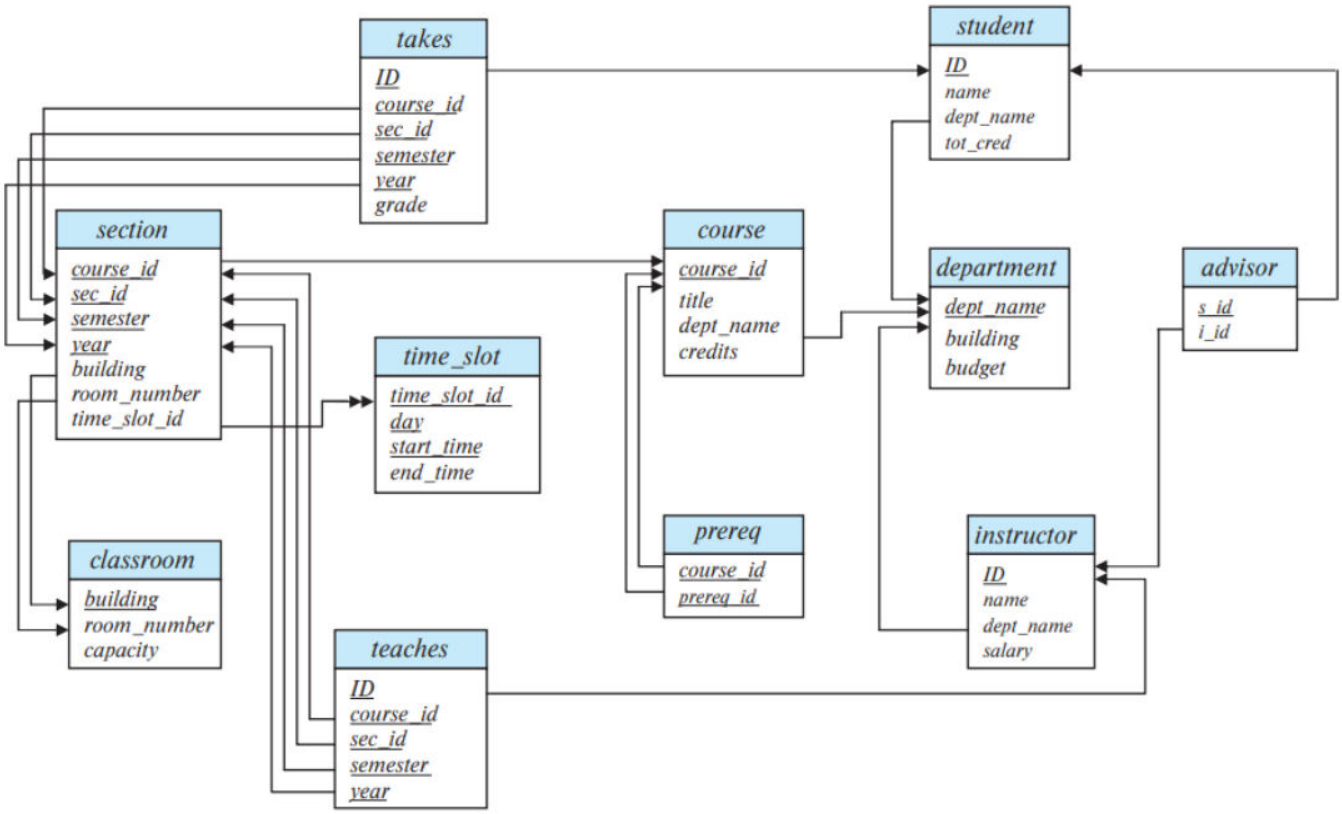
Database Engine (IMPORTANT):

1. Storage manager:
    - Program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
    - It is responsible for interaction with the OS file manager, efficient storing, retrieving and updating of data.

2. Query Processing:
    - A translation of high-level queries into low-level expressions
    - Parsing and translation
    - Optimization
    - Evaluation

3. Transaction Management:
    - Transaction management component: ensures that the database remains in a consistent state despite system failures
    - Concurrency control manager: controls the interaction among the concurrent transactions to ensure the consistency of the database

Introduction to Relational Model



Attribute:

- Piece of data that describes an entity, columns
- Domain of the attribute: set of allowed values for each attribute
- Attribute values need to be atomic (indivisible)
- Null: special value of every domain that indicates that the value is unknown, can cause complications in the definitions of many operations.

Keys (IMPORTANT):

- Superkey: an attribute or set of attributes that can be used to uniquely identify all attributes in a relation. All Superkeys can't be candidate keys. In a relation the no. of Superkeys is always greater than or equal to the no. Candidate Keys. (2^n-1 + 2^n-3 – 2^4)
- Candidate Key: a minimal subset of a Superkey, allowed to be null. Every candidatekey is a superkey
- Primary Key: selected from candidate keys, used to uniquely identify all attributes in a relation.
- Secondary Key/ Alternate Key: all Candidate Keys not selected as a Primary Key
- Surrogate Key/ Synthetic Key: a key that uniquely identifies an entity/ object in the database that is not directly derived from the application data. Example: serial number
- Foreign Key: used to establish relationships between two tables, value in one relation must appear in another
- Simple Key: consists of a single attribute
- Composite Key: more than one attribute, each component not a simple key
- Compound Key: more than one attribute, each component is a simple key

Relational Query Languages:

- Procedural programming: tell the computer how to get the output
- Declarative programming: requires a more descriptive style

Relation Operators:

- Select Operation (σ): selection of rows (where clause)
- Projection Operation (π): selection of columns (select clause)
- Union Operation (∪): union of two relations (or), must have same number of columns, each column must have the same data type
- Difference (-): r – s, what's there in r but not in s, must satisfy same conditions as union
- Intersection (∩): common attributes (and), must satisfy same conditions as union [r∩s = r – (r – s)]
- Cartesian product (×): cross-join which outputs all possible combinations
- Natural Join (⋈): outputs pairs of rows from the two input relations that have the same value on all attributes that have the same name.

Aggregation Operators:

- SUM()
- AVG()
- MAX()
- MIN()

Symbols:

∨ - or

∧ - and

¬ - not

Introduction to SQL

Domain Types in SQL:

- char(n): fixed length character string, with user-specified length n
- varchar(n): variable length character strings, with user-specified maximum length n
- int: integer (a finite subset of the integers that is machine-dependent)
- smallint(n): small integer (a machine-dependent subset of the integer domain type)
- numeric (p, d): fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point. (ex., numeric (3, 1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- real, double precision: floating point and double-precision floating point numbers, with machine-dependent precision
- float(n). Floating point number, with user-specified precision of at least n digits

Create Table Construct (DDL):

create table *instructor* (ID **char**(5), name **varchar**(20) **not null**, dept_name **varchar**(20), salary **numeric**(8,2), **primary key** (ID), **foreign key** (dept_name) **references** department)

- **primary key** declaration on an attribute automatically ensures **not null**

Basic Query Structure (DML):

select A1, A2, An (list of attributes)

from r1, r2, rm (relation)

where P (predicate)

- The result of an SQL query is a relation
- SQL names are case sensitive
- SQL allows duplicates in relations as well as in query results

Select Clause:

- List of attributes
- **distinct** is used after select to eliminate duplicates
- **all** after select is used to keep duplicates
- ***** after select clause clause denotes all attributes
- Can use arithmetic expressions in select
- Can rename resulted query with **as** (**select** ID, name, salary/12 **as** monthly_salary)

From Clause:

- Lists the relations involved in the query
- Cartesian product in relational algebra

Where Clause:

- Specifies conditions that the result must satisfy
- Selection predicate in relational algebra

❖ **select, from, group by, having, order by**

Basic Operations:

- String Operations: (%) matches any substring, represents zero, one, or many characters;
  (_) matches any character, represents one single character; case sensitive
  Examples:
  'Intro%' matches any string beginning with "Intro"
  '%meow' matches any string ending with "meow"
  '%Comp%' matches any string containing "Comp" as a substring

'_ _ _ ' matches any string of exactly three characters
'_ _ _ %' matches any string of at least three characters

- Ordering the Display of Tuples: **order by**, can specify desc or asc. Asc is the default. Can sort on multiple attributes
- **Select top:** specify the number of records to return
- **in** allows for multiple values in a where clause, example: **where** dept_name **in** ('CS', 'Bio)

Set Operations:

All of these operations automatically eliminate duplicates, to keep duplicates use **all** after the operation

- **union** (or)
- **intersect** (and)
- **except** (difference)

Null Values:

- *null* signifies an unknown value or that a value doesn't exist
- **is null/ is not null** is used to check for null values (in where clause)
- result of any arithmetic expression with null is *null*
- any comparison with null returns *unknown*
- logic using *unknown:*
    1. OR: (unknown **or** true) = true
       (unknown **or** false) = unknown
       (unknown **or** unknown) = unknown
    2. AND: (unknown **and** true) = unknown
       (unknown **and** false) = false
       (unknown **and** unknown) = unknown
    3. NOT: (not unknown) = unknown

Aggregate Functions:

These values operate on the multiset of values of a column of a relation and return a value

All aggregate operations except **count(*)** ignores tuples with null values

- **avg**: average value
- **min**: minimum value
- **max**: maximum value
- **sum**: sum of values
- **count**: number of values
- **group by**: form groups
- **having**: used after forming groups, like where clause for groups

Examples:

1. Select distinct:
   From the classroom relation find the names of buildings in which every individual classroom has capacity less than 100 (removing the duplicates).

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

select distinct building

from classroom

where capacity < 100

2. Select all (duplicate retention is default):
   From the classroom relation in the figure, find the names of buildings in which every individual classroom has capacity more than 100 (without removing the duplicates).

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

select all building

from classroom

where capacity > 100

3. Where Clause **and/or**:
   From the instructor and department relations in the figure, find out the names of all instructors whose department is Finance or whose department is in any of the following buildings: Watson, Taylor.

   select i.name
   from instructors i, department d
   where d.dept_name = i.dept_name **and**
   (d.dept_name = 'Finance' **or** d.building **in** ('Watson', 'Taylor'))

4. String Operations:
   From the course relation, find the titles of all courses whose course id has three alphabets indicating the department.

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

select title
from course
where course_id like '_ _ _-%'

5. Order by:
   From the student relation in the figure, obtain the list of all students in alphabetic order of departments and within each department, in decreasing order of total credits.

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

select name, dept_name, tot_cred
from student
order by dept_name **asc**, tot_cred **desc**

6. In Operator:
   From the teaches relation, find the IDs of all courses taught in the Fall or Spring of 2018.

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

select *course_id*
from *teaches*
where *semester* **in** *('Fall','Spring')*
  and *year*=2018;

7. Union
   From the teaches relation, find the IDs of all courses taught in the Fall or Spring of 2018.

   select course id
   from teaches
   where semester='Fall' **and** year=2018
   union
   select course id
   from teaches
   where semester='Spring' **and** year=2018

8. Intersect:
   From the instructor relation, find the names of all instructors who taught in either the Computer Science department or the Finance department and whose salary is < 80000

select name
from instructor
where dept name **in** ('Comp. Sci.','Finance')
intersect
select name
from instructor
where salary < 80000

9. Except:
   From the instructor relation, find the names of all instructors who taught in either the Computer Science department or the Finance department and whose salary is either ≥ 90000 or ≤ 70000

   select name
   from instructor
   where dept name in ('Comp. Sci.','Finance')
   except
   select name
   from instructor
   where salary < 90000 and salary > 70000

10. Avg:
    From the classroom relation, find the names and the average capacity of each building whose average capacity is greater than 25.

    select building, avg (capacity)
    from classroom
    group by building
    having avg (capacity) > 25

11. Min:
    From the instructor relation, find the least salary drawn by any instructor among all the instructors

    select min(salary) **as** least salary
    from instructor

12. Max:
    From the student relation, find the maximum credits obtained by any student among all the students

    select max(tot cred) as max credits
    from student

13. Count:
    From the section relation, find the number of courses run in each building

    select building, **count**(course id) **as** course count
    from section
    group by building

14. Sum:
    From the course relation, find the total credits offered by each department

select dept name, **sum**(credits) **as** sum credits
from course
group by dept name

Intermediate SQL

Nested Subqueries:

- A subquery is a select-from-where expression that is nested within another query
- Can be used in select, from or where clause

Subqueries in the Where clause:

Examples:

- Find courses offered in Fall 2009 **and** in Spring 2010 (intersect)

  select distinct course_id
  from section
  where semester = 'Fall' **and** year = '2009' **and** course_id **in**
  (**select** course_id
  from section
  where semester = 'Spring' **and** year = '2010')

  easier way to solve with set operations:

  select distinct course_id
  from section
  where semester = 'Fall' **and** year = '2009'
  intersect
  select distinct course_id
  from section
  where semester = 'Spring' **and** year = '2010'

- Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101 (in)

  select count (distinct ID)
  from takes
  where (course_id, sec_id, semester, year) **in**
  (**select** course_id, sec_id, semester, year
  from teaches
  where teaches.ID = 10101)

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department (some)

  select name
  from instructor
  where salary > **some** (**select** salary

from instructor
where dept_name = 'Biology')

not using **some** clause

select distinct i.name
from instructor i, instructor t #self join
where i.salary > t.salary **and** t.dept_name = 'Biology'

some represents *existential quantification*

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$) = true (read: 5 < some tuple in the relation)

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = false

(5 = **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true

(5 ≠ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department (all)

  select name
  from instructor
  where salary > **all** (**select** salary
  from instructor
  where dept_name = 'Biology')

  all represents *universal quantification*

(5 < **all** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$) = false

(5 < **all** $\begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline \end{array}$) = true

(5 = **all** $\begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline \end{array}$) = false

(5 ≠ **all** $\begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline \end{array}$) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**
However, (= **all**) ≢ **in**

- Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester (exists)

  select course_id
  from section s
  where semester = 'Fall' and year = 2009 and **exists**
  (**select ***

```
from section t
where semester = 'Spring' and year = 2010
and s.course_id = t.course_id)
```

- Find all students who have taken all courses offered in the Biology department (not exists)
  Cannot write this using all and its variants

```
select distinct s.ID, s.name
from student s
where not exists ((select course_id
                   from course
                   where dept_name = 'Biology')
                   except
                   (select t.course_id
                    from takes as t
                    where s.ID = t.ID))
```

- Find all courses that were offered at most once in 2009 (unique)
  The unique constructs tests whether a subquery has any duplicates tuples in its result
  It evaluates to true if a given subquery has no duplicates

```
select c.course_id
from course c
where unique (select s.course_id
              from section s
              where c.course_id = s.course_id and s.year = 2009)
```

Subqueries in the From Clause:

Examples:

- Find the average instructors' salaries of those departments where the average salary is greater than $42,000

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

- The **with** clause provides a way of defining a temporary relation whose definition is only available to the query in which the with clause occurs
  Find all departments with the maximum budget

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

  Find all departments where the total salary is greater than the average of the total salary of all departments

```
with dept_total(dept_name, value) as
```

```
                    select dept_name, sum(salary)
                    from instructor
                    group by dept_name,
        dept_total_avg(value) as
                    (select avg(value)
                    (from dept_total)
    select dept_name
    from dept_total, dept_total_avg
    where dept_total.value > dept_total_avg.value
```

Subqueries in the Select Clause:

Scalar subquery is one which is used where a single value is expected

- List all departments along with the number of instructors in each department

```
select dept_name, (select count(*)
                    from instructor, department
                    where department.dept_name = instructor.dept_name)
as num_instructors
from department
```

Modifications of the Database:

Deletion:

Deletion of tuples from a given relation

Format:

- **delete from** r1, r2..rm 2
  where P

Examples:

- Delete all instructors

```
delete from instructor
```

- Delete all instructors from the Finance department

```
delete from instructor
where dept_name = 'Finance'
```

- Delete all instructors whose salary is less than the average salary of instructors

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

  Problem: as we delete tuples from deposit, the average salary changes
  The correct solution will use the with clause to compute and store the avg salary and then find and delete all the tuples with salary less than avg

Insertion:

Inserting new tuples into a relation

Format:

- **insert into** r1, r2
  values x1,x2

Examples:

- Add a new tuple to course
  insert into course
  values ('MEW-101, 'Cat Science', 'Cat. Sci.', 4)

- Add all instructors to the student relation with tot creds set to 0
  insert into student
  select ID, name, dept_name, 0
  from instructor

Updating:

Updating values in some tuples of a relation

Format:

- **update** r1
  set values
  where P

Examples:

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  update instructor
  set salary = salary * 1.03
  where salary > 100000
  update instructor
  set salary = salary * 1.05
  where salary <= 100000

Order is important

update instructor
set salary = case
when salary <= 100000
then salary * 1.03
else salary * 1.05
end

Join Expressions:

- Join operations take two relations and return as a result another relation
- A join operation is a Cartesian product which requires that tuples in the two relations matches (under some condition).
- It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the from clause

Cross Join:

- Returns the cartesian product of rows from tables in the join
  no. of tuples returned = m * n (without condition)
- Examples:

```
select *
from employee cross join department

select *
from employee, department
```

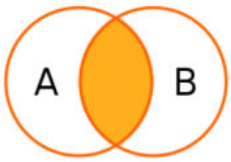
Inner Join:

- Joins two table on the basis of the column which is explicitly specified in the ON clause with a given condition using a comparison operator

```
course inner join prereq on
course.course_id = prereq.course_id
```

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101



course_id	title	dept_name	credits	prere_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- If it is natural join (joins two tables based on same attribute name and compatible datatypes) the duplicate columns are skipped

```
course natural inner join prereq on
course.course id = prereq.course id
```

course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

Outer Join:

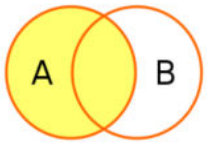
- An extension of the join operation that avoids loss of information
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join
- Uses null values

Left Outer Join:

- Includes all tuples from relation A, if attributes not correspondingly in B, B gets a null value
  course **natural left outer join** prereq **on**
  course.course id = prereq.course id

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101



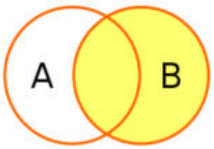
course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Outer Join:

- Includes all tuples from relation B, if attributes not correspondingly in A, A gets a null value
  course **natural right outer join** prereq **on**
  course.course id = prereq.course id

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101



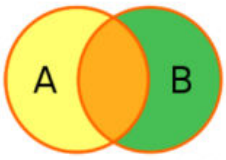
course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Full Outer Join:

- Tuples from A&B are preserved
  course **full outer join** prereq **using** (course id)

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101



course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Views:

- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a view.
- A view provides a mechanism to hide certain data from the view of certain users
- Doesn't create a new relation, virtual, isn't stored

- A view can be used to create another view
- Modifying the data will not affect the view (unless it's a simple view)
- Views need to be maintained by updating the view
- Materializing a view - create a physical table containing all the tuples in the result of the query defining the view
- Format:
  create view v **as**
  <any legal SQL statement>
- Example:
  A view of instructors without their salary

  create view *faculty* **as**
  select ID, name, dept name
  from instructor

  Updating this view

  insert into *faculty*
  values ('30765', 'Green', 'Music')

Integrity Constraints:

Guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency

- **not null:** specified value can't be null
- **unique(a1, a2..am):** a1, a2, am form a candidate key
- **check(P): check** (semester in ('Fall', 'Winter', 'Spring', 'Summer'))

Referential Integrity:

- ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
- Example: If 'Biology' is a department name appearing in one of the tuples in the instructor relation, then there exists a tuple in the department relation for 'Biology'.
- The Cascading Referential Integrity Constraints in SQL Server are the foreign key constraints that tell SQL Server to perform certain actions whenever a user attempts to delete or update a primary key to which an existing foreign keys point.

SQL Data Types and Schemas:

- Date: 'yyyy-mm-dd'
- Time: '09:00:30'/ '09:00:30.75'
- Timestamp: date + time
- Interval: period of time
- User defined types [**create type** Dollars **as numeric**(12,2) **final**]
- blob (binary large object)
- clob (character large object)
- User defined domains [**create domain** person_name **char**(20) **not null**]

Authorization:

- Forms of authorization to modify the database schema:
  Index - allows creation and deletion of indices
  Resources - allows creation of new relations
  Alteration - allows addition or deletion of attributes in a relation
  Drop - allows deletion of relations

- Forms of authorization on parts of the database:
  Read - allows reading, but not modification of data
  Select – allows read access and ability to query a view
  Insert - allows insertion of new data, but not modification of existing data
  Update - allows modification, but not deletion of data
  Delete - allows deletion of data

- The grant statement is used to confer authorization
  grant <privilege list>
  on <relation name or view name>
  to <user list>
  Granting a privilege on a view does not imply granting any privileges on the underlying relations.
  The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator)

- The revoke statement is used to revoke authorization
  revoke <privilege list>
  on <relation name or view name>
  from <user list>
  If includes public, all users lose the privilege except those granted it explicitly
  If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation
  All privileges that depend on the privilege being revoked are also revoked

- Roles: roles can be granted to users, as well as to other roles
  Create role-
  create role instructor
  grant instructor **to** Amit
  Privileges can be granted to roles-
  grant select on takes **to** instructor
  Roles can be granted to users, as well as to other roles-
  create role teaching assistant
  grant teaching assistant **to** instructor
  There can be a chain of roles

Advanced SQL

SQL Functions:

Function is a tool in SQL that is used to calculate anything to produce an output for the provided inputs

- Define a function that, given the name of a department, returns the count of the number of instructors in that department

```
create function dept_count (dept_name varchar(20))
returns integer #indicate variable type that is return
begin
declare d_count integer
        select count (*) into d_count
        from instructor
        where instructor.dept name = dept_name
return d_count #what to return
end
```

  The function dept count can be used to find the department names and budget of all departments with more that 12 instructors:

```
select dept name, budget
from department
where dept count (dept name ) > 12
```

- Table functions: functions that return a relation as a result
  Return all instructors in a given department

```
create function instructor of (dept_name char(20))
returns table ( ID varchar(5), name varchar(20), dept_name varchar(20) salary numeric(8, 2))
return table (select ID, name, dept_name, salary
                from instructor
                where instructor.dept_name = instructor_of.dept_name)
```

Procedures:

A procedure is a set of instructions which takes input and performs a certain task

- The dept count function could instead be written as procedure

```
create procedure dept_count_proc (in dept_name varchar (20), out d count integer)
begin
select count(*) into d_count
from instructor
where instructor.dept_name = dept_count_proc.dept_name
end
```

❖ Language constructs for procedures and functions - while loop, repeat loop, for loop, if-then-else, case

Differences between function and procedure:

Key	Function	Procedure
Definition	A function is used to calculate result using given inputs.	A procedure is used to perform certain task in order.
Call	A function can be called by a procedure.	A procedure cannot be called by a function.
DML	DML statements cannot be executed within a function.	DML statements can be executed within a procedure.
SQL, Query	A function can be called within a query.	A procedure cannot be called within a query.
SQL, Call	Whenever a function is called, it is first compiled before being called.	A procedure is compiled once and can be called multiple times without being compiled.
SQL, Return	A function returns a value and control to calling function or code.	A procedure returns the control but not any value to calling function or code.
try-catch	A function has no support for try-catch	A procedure has support for try-catch blocks.
SELECT	A select statement can have a function call.	A select statement can't have a procedure call.
Explicit Transaction Handling	A function cannot have explicit transaction handling.	A procedure can use explicit transaction handling.

Triggers:

- A trigger defines a set of actions that are performed in response to an insert, update, or delete operation on a specified table.
  When such an SQL operation is executed, the trigger is said to have been activated
  Triggers are optional
  Triggers are defined using the create trigger statement
- Triggers can be used:
  to enforce data integrity rules via referential constraints and check constraints
  to cause updates to other tables, automatically generate or transform values for inserted or updated rows
  invoke functions to perform tasks such as issuing alerts
- To design a trigger mechanism the following must be done:
  specify the events / (like update, insert, or delete) for the trigger to executed
  specify the time (BEFORE or AFTER) of execution
  specify the actions to be taken when the trigger executes

1. BEFORE triggers

- Run before an update, or insert
- Values that are being updated or inserted can be modified before the database is actually modified.
- You can use triggers that run before an update or insert to: Check or modify values before they are actually updated or inserted in the database − Useful if user-view and internal database format differs. Run other non-database operations coded in user-defined functions

2. BEFORE DELETE triggers

- Run before a delete
- Checks values (raises an error, if necessary)

3. AFTER triggers

- Run after an update, insert, or delete
- You can use triggers that run after an update or insert to: Update data in other tables − Useful for maintain relationships between data or keep audit trail . Check against other data in the table or in other tables − Useful to ensure data integrity when referential integrity constraints aren't appropriate, or − when table check constraints limit checking to the current table only. Run non-database operations coded in user-defined functions − Useful when issuing alerts or to update information outside the database

- **Row level triggers** are executed whenever a row is affected by the event on which the trigger is defined, fires once for each row affected.
- **Statement level triggers** perform a single action for all rows affected by a statement, instead of executing a separate action for each affected row, fires once per triggering event.
- Triggering event can be an insert, delete or update

Uses:

- Logging changes to a history table
- Auditing users and their actions against sensitive tables
- Simple validation

Formal Relational Query Languages

Relational Algebra:

- Procedural and Algebra based
- The operators take one or two relations as inputs and produce a new relation as a result
- Six basic operators
    - select: σ
    - project: Π
    - union: ∪
    - set difference: −
    - Cartesian product: x
    - rename: ρ

- Symbols:
  - $\vee$ or
  - $\wedge$ and
  - $\neg$ not

- Select Operation (σ): selection of rows (where clause)
- Projection Operation (π): selection of columns (select clause)
- Union Operation ($\cup$): union of two relations (or), must have same number of columns, each column must have the same data type
- Difference (-): r – s, what's there in r but not in s, must satisfy same conditions as union
- Intersection ($\cap$): common attributes (and), must satisfy same conditions as union [$r \cap s = r - (r - s)$]
- Cartesian product ($\times$): cross-join which outputs all possible combinations
- Natural Join ($\bowtie$): outputs pairs of rows from the two input relations that have the same value on all attributes that have the same name

- Examples:
  - $\sigma_{dept\_name\ =\ 'Physics'}(instructor)$

    where dept_name = 'Physics'
  - $\Pi_{ID,name,salary}(instructor)$

    select ID, name, salary **from** instructor
  - $\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2009}(section)) \cup \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2010}(section))$

    select course_id
    from section
    where semester = 'Fall' **and** year = '2009'
    union
    select course_id
    from section
    where semester = 'Spring' **and** year = '2010'
  - $\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2009}(section)) - \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2010}(section))$

    select course_id
    from section
    where semester = 'Fall' **and** year = '2009'
    except
    select course_id
    from section
    where semester = 'Spring' **and** year = '2010'

- Division Operator (÷):
  - Division is a derived operation and can be expressed in terms of other operations

    $r \div s \equiv \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$
  - Examples:

• R

Lecturer	Module
Brown	Compilers
Brown	Databases
Green	Prolog
Green	Databases
Lewis	Prolog
Smith	Databases

S

Subject
Prolog

R | S

Lecturer
Green
Lewis

• R

Lecturer	Module
Brown	Compilers
Brown	Databases
Green	Prolog
Green	Databases
Lewis	Prolog
Smith	Databases

S

Subject
Databases
Prolog

R | S

Lecturer
Green

A

sno	*pno*
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

A	*B*	*C*	*D*	*E*
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

r

D	*E*
a	1
b	1

s

• $r \div s$:

A	*B*	*C*
α	a	γ
γ	a	γ

Tuple Relational Calculus:

- Non-Procedural and Predicate Calculus based
- Each query is of the form {t | P(t)}, t is resulting tuples, P(t) is the predicate used to fetch t.
- Predicate calculus formula:
  - Set of attributes and constants
  - Set of comparison operators: (e.g., , ≥)
  - Set of connectives: and (∧), or (∨), not (¬) d)
  - Implication (⇒) : x ⇒ y, if x if true, then y is true x ⇒ y ≡ ¬ x ∨ y
  - Set of quantifiers: • ∃t ∈ r (Q(t)) ≡ "there exists" a tuple in t in relation r such that predicate Q(t) is true [existential quantifier]
  - ∀t ∈ r (Q(t)) ≡ Q is true "for all" tuples t in relation r [universal quantifier]

- Symbols:
  - ⇒ implies
  - ∃ there exists
  - ∈ belonging to
  - ≡ is defined as
  - ∀ for all
  - ∧ and
  - ∨ or
  - ¬ not

- Examples:
  - **Student**

Fname	Lname	Age	Course
David	Sharma	27	DBMS
Aaron	Lilly	17	JAVA
Sahil	Khan	19	Python
Sachin	Rao	20	DBMS
Varun	George	23	JAVA
Simi	Verma	22	JAVA

    Obtain the first name of students whose age is greater than 21
    {t | ∃s ∈ Student(s.age > 21 ∧ t.Fname = s.Fname)}

  - **student**(rollNo, name, year, courseId)
    course(courseId, cname,teacher)

    1. Find out the names of all students who have taken the course name 'DBMS'

    {t | ∃s ∈ student ∃c ∈ course(s.courseId = c.courseId ∧ c.cname = 'DBMS' ∧ t.name = s.name)}
    {s.name | s ∈ student ∧ ∃c ∈ course(s.courseId = c.courseId ∧ c.cname= 'DBMS')}

    2. Find out the names of all students and their rollNo who have taken the course name 'DBMS'

    {t | ∃s ∈ student ∃c ∈ course(s.courseId = c.courseId ∧ c.cname ='DBMS' ∧ t.name = s.name ∧ t.rollNo = s.rollNo)}
    {s.name, s.rollNo | s ∈ student ∧ ∃c ∈ course(s.courseId = c.courseId ∧ c.cname = 'DBMS')}

- Relational Algebra (RA) vs Tuple Relational Calculus (TRC)

Flights(flno, from, to, distance, departs, arrives)
Aircraft(aid, aname, cruisingrange)
Certified(eid, aid)
Employees(eid, ename, salary)

Find the eids of certified pilots working on Boeing aircrafts

RA

$$\Pi_{eid}(\sigma_{aname='Boeing'}(Aircraft \ltimes Certified))$$

TRC

- $\{C.eid \mid C \in Certified \wedge \exists A \in Aircraft(A.aid = C.aid \wedge A.aname = \text{'Boeing'})\}$
- $\{T \mid \exists C \in Certified \exists A \in Aircraft(A.aid = C.aid \wedge A.aname = \text{'Boeing'} \wedge T.eid = C.eid)\}$

Domain Relational Calculus:

- Non-Procedural and Predicate Calculus based
- Each query is an expression of the form: $\{< x1, x2, \ldots, xn > | P(x1, x2, \ldots, xn)\}$
  x1, x2, . . . , xn represent domain variables
  P represents a formula similar to that of the predicate calculus
- Example: return the name and age of students having marks above 75
  $\{< a, b > | \exists a, b, c, d(< a, b, c, d > \in \text{students} \wedge c > 75)\}$

RA vs TRC vs DRC:

Select Operation

R = (A, B)

Relational Algebra: $\sigma_{B=17}(r)$

Tuple Calculus: $\{t \mid t \in r \wedge B = 17\}$

Domain Calculus: $\{<a, b> \mid <a, b> \in r \wedge b = 17\}$

Project Operation

R = (A, B)

Relational Algebra: $\Pi_A(r)$

Tuple Calculus: $\{t \mid \exists\ p \in r\ (t[A] = p[A])\}$

Domain Calculus: $\{<a> \mid \exists\ b\ (<a, b> \in r)\}$

Combining Operations

R = (A, B)

Relational Algebra: $\Pi_A(\sigma_{B=17}(r))$

Tuple Calculus: $\{t \mid \exists\, p \in r\ (t[A] = p[A] \wedge p[B] = 17)\}$

Domain Calculus: $\{<a> \mid \exists\, b\ (<a, b> \in r \wedge b = 17)\}$

Union

R = (A, B, C) S = (A, B, C)

Relational Algebra: $r \cup s$

Tuple Calculus: $\{t \mid t \in r \vee t \in s\}$

Domain Calculus: $\{<a, b, c> \mid <a, b, c> \in r \vee <a, b, c> \in s\}$

Set Difference

R = (A, B, C) S = (A, B, C)

Relational Algebra: $r - s$

Tuple Calculus: $\{t \mid t \in r \wedge t \notin s\}$

Domain Calculus: $\{<a, b, c> \mid <a, b, c> \in r \wedge <a, b, c> \notin s\}$

Intersection

R = (A, B, C) S = (A, B, C)

Relational Algebra: $r \cap s$

Tuple Calculus: $\{t \mid t \in r \wedge t \in s\}$

Domain Calculus: $\{<a, b, c> \mid <a, b, c> \in r \wedge <a, b, c> \in s\}$

Cartesian/Cross Product

R = (A, B) S = (C, D)

Relational Algebra: $r \times s$

Tuple Calculus: $\{t \mid \exists p \in r \exists q \in s (t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = q[C] \wedge t[D] = q[D])\}$

Domain Calculus: $\{\langle a, b, c, d\rangle \mid \langle a, b\rangle \in r \wedge \langle c, d\rangle \in s\}$

Natural Join

R = (A, B, C, D) S = (B, D, E)

Relational Algebra: $r \bowtie s$

$\Pi_{r.A,r.B,r.C,r.D,s.E}(\sigma_{r.B=s.B \wedge r.D=s.D} (r \times s))$

Tuple Calculus: $\{t \mid \exists p \in r \exists q \in s (t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = p[C] \wedge t[D] = p[D] \wedge t[E] = q[E] \wedge p[B] = q[B] \wedge p[D] = q[D])\}$

Domain Calculus: $\{\langle a, b, c, d, e\rangle \mid \langle a, b, c, d\rangle \in r \wedge \langle b, d, e\rangle \in s\}$

Division

R = (A, B) S = (B)

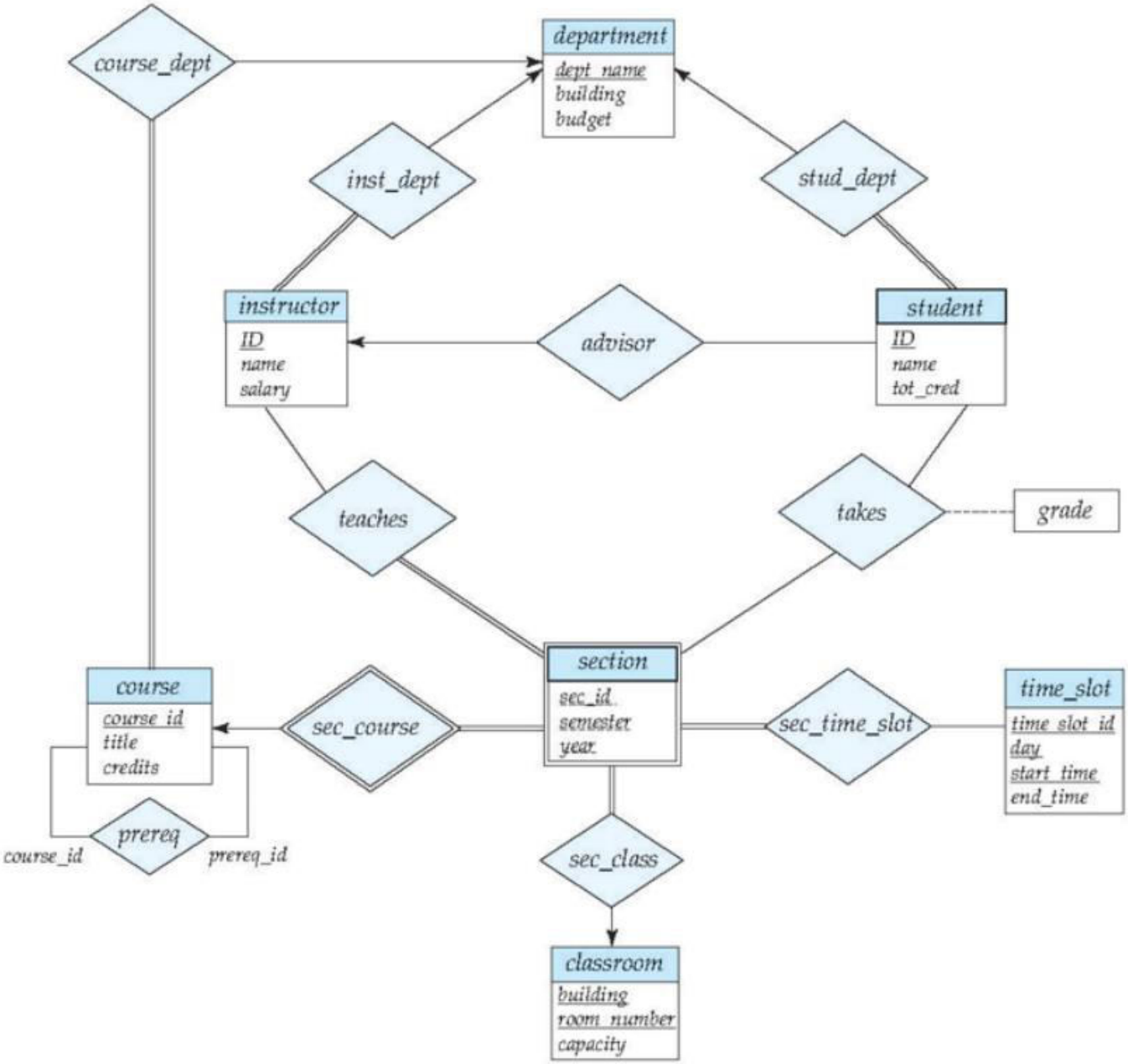
Relational Algebra: $r \div s$

Tuple Calculus: $\{t \mid \exists p \in r \forall q \in s (p[B] = q[B] \Rightarrow t[A] = p[A])\}$

Domain Calculus: $\{\langle a\rangle \mid \langle a\rangle \in r \wedge \forall \langle b\rangle (\langle b\rangle \in s \Rightarrow \langle a, b\rangle \in r)\}$

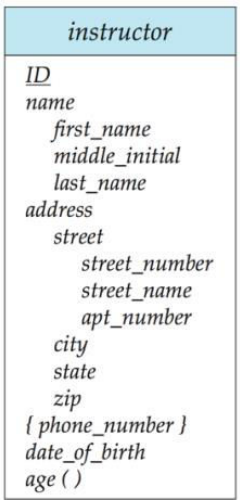
Entity- Relationship Model

- The ER data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database
- The ER model is useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema
- ER Diagram for a University Enterprise:

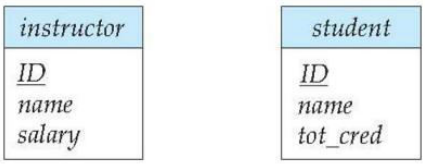


Attributes:

- An Attribute is a property associated with and entity / entity set. Based on the values of certain attributes, an entity can be identified uniquely
- **Domain**: Set of permitted values for each attribute
- Attribute types:
    - **Simple**: cannot be further divided into sub attributes, eg: roll_no
      Composite attributes: can be divided into sub attributes, eg: name [first_name, last_name]
    - **Single-valued:** can only take a single value, eg: ID student can only have one ID
      Multivalued attributes: can take many values, eg: email can have multiple emails
    - **Derived attributes:** computed from other attributes, eg: age
    - **Complex attributes:** those attributes, which can be formed by the nesting of composite and multi-valued attributes, notation to express entity with complex attributes:



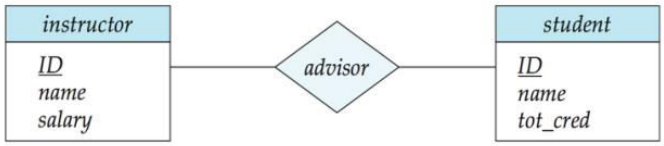
Entity: an object that exists and is distinguishable from other objects. An entity is represented by a set of attributes



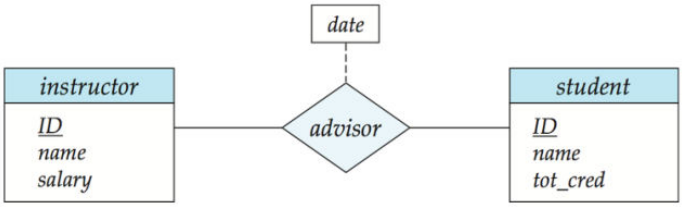
Relationship: an association among several entities

Relationship sets:

- **Relationship set**: a mathematical relation between 2 or more entities, each taken from entity sets



- An attribute can also be associated with a relationship set.
  For instance, the advisor relationship set between entity sets instructor and student may have the attribute date which tracks when the student started being associated with the advisor.



- **Binary relationship:** involves two entity sets (or degree two). Most relationship sets in a database system are binary

Entity Sets:

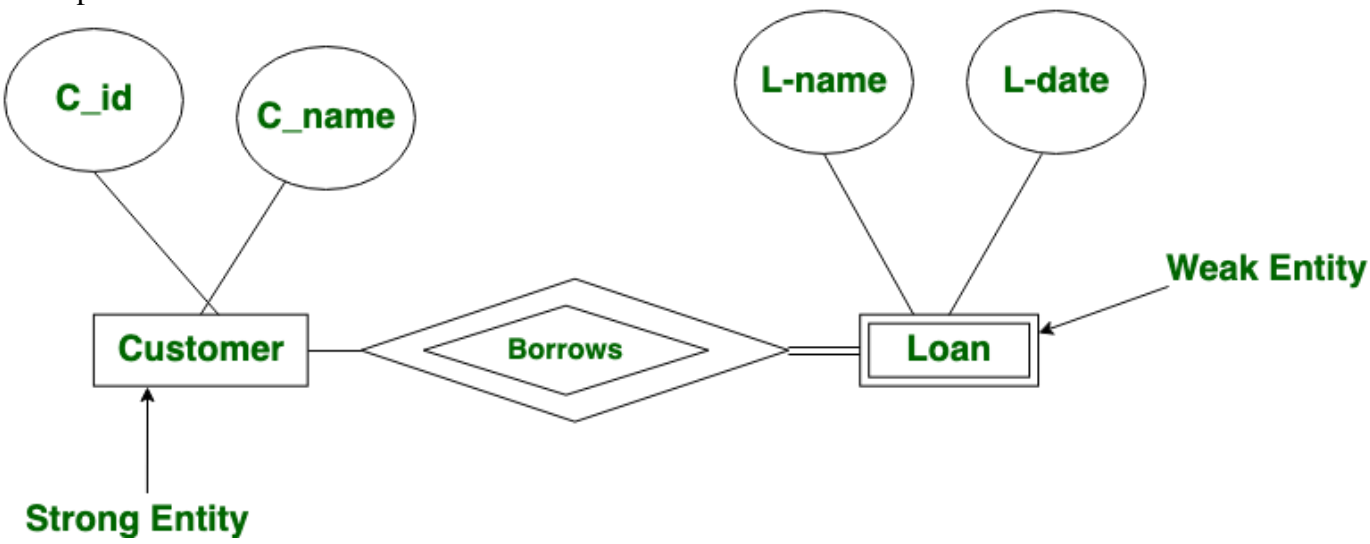
- **Entity set**: a set of entities of the same type that share the same properties
- **Primary key**: a subset of the attributes form a primary key of the entity set; that is, uniquely identifying each member of the set. Underline indicates primary key attributes.
- Rectangles represent entity sets. Attributes are listed inside entity rectangle.

Strong Entity Sets:

- A strong entity set **has** a primary key to uniquely identify all its entities
- Primary key of a strong entity set is represented by underlining it

Weak Entity Sets:

- A weak entity set **doesn't** have a primary key to uniquely identify all its entities
- **Discriminator:** can identify a group of entities from the entity set. Represented by underlining with a dashed line
- Since a weak entity set does not have primary key, it cannot independently exist in the ER Model. It features in the model in relationship with a strong entity set. This is called the **identifying relationship**.
- **Primary Key of Weak Entity Set** = Its own discriminator + Primary Key of Strong Entity Set
- Weak entity is represented by **double rectangle**
- Weak entity set must have **total participation** in the identifying relationship. That is all its entities must feature in the relationship
- Total participation is indicated by **double line**
- Example:



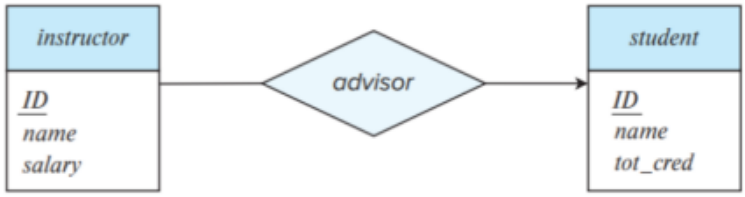
- Loan as the weak entity has total participation, each loan must have a customer.
- Customer as the strong entity has partial participation, each customer doesn't need to have a loan.

Cardinality Constraints:

- Express the number of entities to which another entity can be associated via a relationship set
- **One-to-many**: an instructor is associated with several (including 0) students via advisor, a student is associated with at most one instructor via advisor

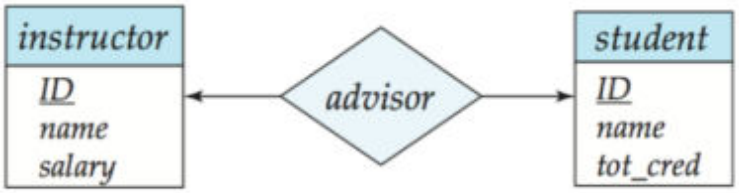
one-to-many relationship

- **Many-to-one**: an instructor is associated with at most one student via advisor, and a student is associated with several (including 0) instructors via advisor



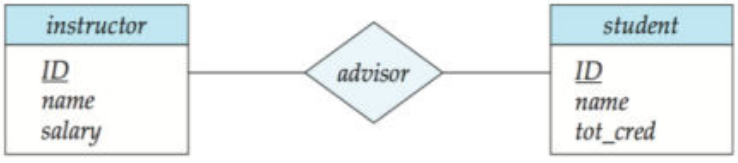
many-to-one relationship

- **One-to-one**: one instructor to one student via advisor



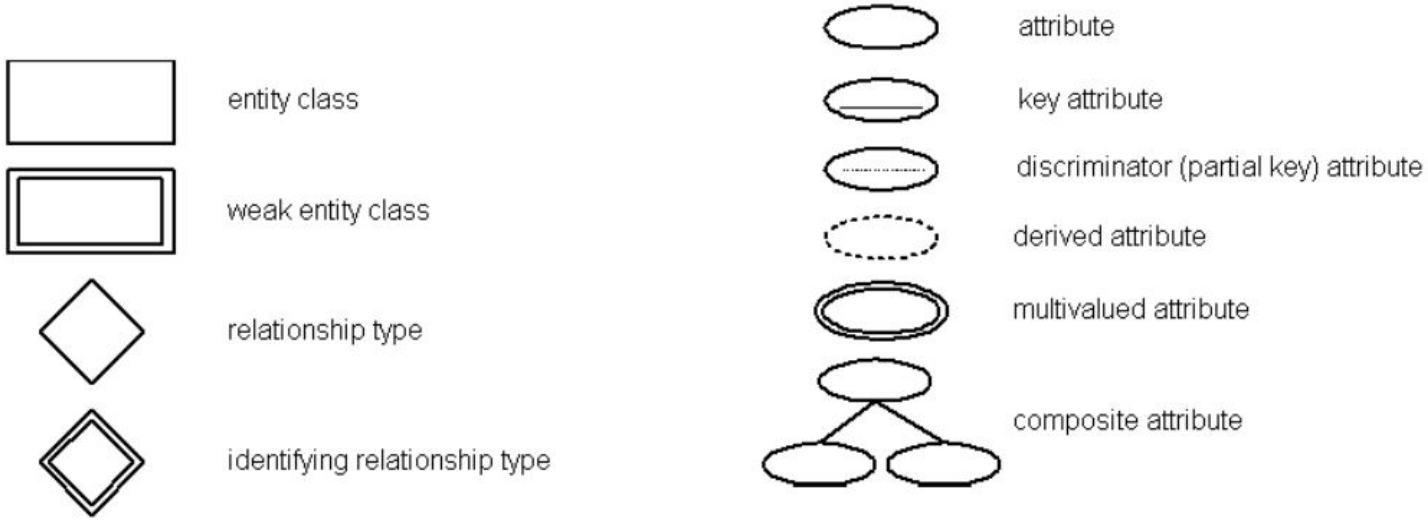
one-to-one relationship

- **Many-to-many:** An instructor is associated with several (possibly 0) students via advisor. A student is associated with several (possibly 0) instructors via advisor



many-to-many relationship

E-R Diagram Symbols:



Translating E-R Diagram into Relational Schema:

Tutorial 4.3 Translating ER Diagram into Relational Schema

Relational Database Design

Good Relational Design:

- Reflects real-world structure of the problem
- Can represent all expected data over time
- Avoids redundant storage of data items
- Provides efficient access to data
- Supports the maintenance of data integrity over time
- Clean, consistent, and easy to understand

❖ Note: These objectives are sometimes contradictory!

Redundancy: having multiple copies of same data in the database.

- This problem arises when a database is not normalized
- It leads to anomalies
- Dependencies create redundancy

Anomaly: inconsistencies that can arise due to data changes in a database with insertion, deletion, and update. These problems occur in poorly planned, un-normalised databases where all the data is stored in one table (a flat-file database).

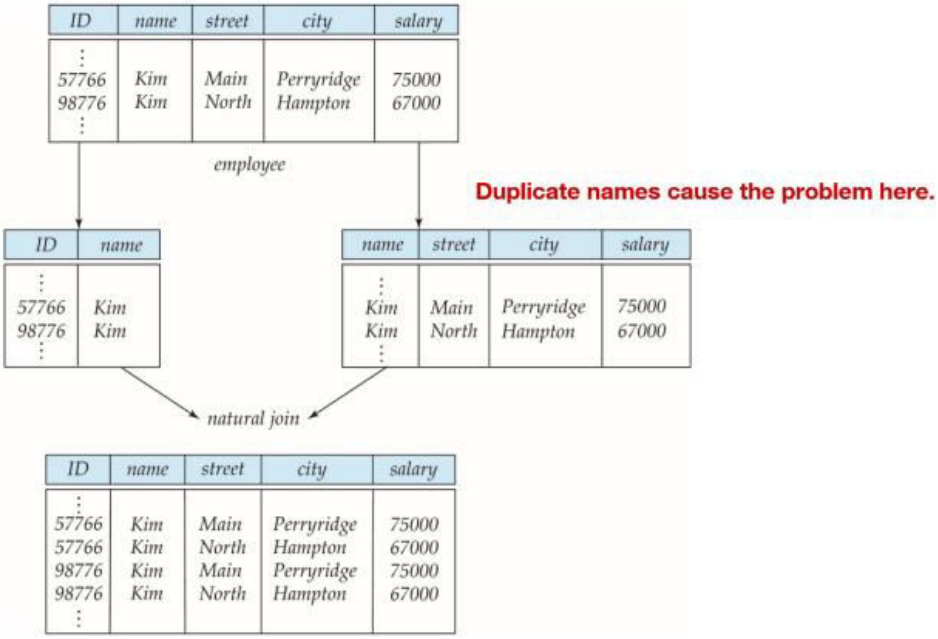
- Insertions Anomaly: when the insertion of a data record is not possible without adding some additional unrelated data to the record
- Deletion Anomaly: when deletion of a data record results in losing some unrelated information that was stored as part of the record that was deleted from a table

- Update Anomaly: when a data is changed, which could involve many records having to be changed, leading to the possibility of some changes being made incorrectly

Decomposition: partitioning a relation into smaller relations

- A good decomposition minimises dependencies
- Various schemes of normalization ensure good decomposition

Lossy decomposition: a reconstruct of the original relation is not possible after decomposition, lost info.



Lossless-Join Decomposition: original relation can be reconstructed by using natural join on sub relations.

Decomposition of $R = (A, B, C)$
$R_1 = (A, B)$, $R_2 = (B, C)$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_{A,B}(r) \bowtie \Pi_{B,C}(r)$

A	B	C
α	1	A
β	2	B

Functional Dependencies (IMPORTANT):

- A functional dependency is a generalization of the notion of a key
- **Definition:** functional dependencies (FD) are constraints on the set of legal relations which require that the value for a certain set of attributes uniquely determine the value for another set of attributes
- SQL does not provide a direct way of specifying functional dependencies other than superkeys

- If **t1[α] = t2[α] ⇒ t1[β] = t2[β]**, then **α → β**
- Example: Consider R(A, B) with the following instance of r

A	B
1	4
1	5
3	7

On this instance, A → B does NOT hold, but B → A does hold
All values in A aren't unique so if you're asked, what is B if A = 1? You will get two values for B (4 and 5).
But
All values in B are unique and determine A, what is A when B is 4? You will get a unique value A (1).
Therefore, B → A (B determines A)

- A functional dependency is trivial if it is satisfied by all instances of a relation
  In general, α → β is trivial if β ⊆ α
  Example: ID, name → ID, name → name

❖ **Armstrong's Axioms:**
Reflexivity (trivial property): if β ⊆ α, then α → β (if AB → B, then A → B)
Augmentation: if α → β, then γα → γβ
Transitivity: if α → β and β → γ, then α → γ

Additional Derived Rules:
Union: if α → β holds and α → γ holds, then α → βγ holds
Decomposition: if α → βγ holds, then α → β holds and α → γ holds
Pseudotransitivity: if α → β holds and γβ → δ holds, then αγ → δ holds

These axioms are **Sound;** given a set of functional dependencies F specified on a relation schema R, any dependency that we can infer from F by using the primary rules of Armstrong axioms holds in every relation state r of R that satisfies the dependencies in F
and **Complete;** using primary rules of Armstrong axioms repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of all possible dependencies that can be inferred from F called the **Closure Set F+** for FDs F.

- **Closure set of FDs (F+):** A set of all FDs implied by the original set of dependencies F
  Example: F = {A → B, B → C}
  F + = {A → B, B → C, A → C}

❖ **Closure of Attribute Sets** (α+)**:** the set of attributes that are functionally determined by α under F, where α is a set of attributes
Example: R = (A, B, C, G, H, I) ; F = {A → B, A → C, CG → H, CG → I, B → H}
We need to find (AG)+

a) result = AG (trivial)
b) result = ABCG (A → C and A → B)
c) result = ABCGH (CG → H and CG ⊆ AGBC)
d) **result = ABCGHI** (CG → I and CG ⊆ AGBCH)

Uses of the attribute closure algorithm:

- To test if α is a superkey, we compute α +, and check if α + contains all attributes of R
- To test functional dependencies
- To compute F+

- **Extraneous attributes**: an attribute of an FD is said to be extraneous if we can remove it without changing F+

- Equivalence of Sets of Functional Dependencies:

Condition	CASES			
F Covers G	True	True	False	False
G Covers F	True	False	True	False
Result	F=G	F⊃G	G⊃F	No Comparison

- **Canonical Cover/ Minimal Cover/ Irreducible Sets:** a minimal set of FDs. Should satisfy the following conditions:
  Equivalent to F (F+ = Fc+)
  No FD in Fc contains an extraneous attribute
  Each left side of FD in Fc is unique. That is, there are no two dependencies α1 → β1 and α2 → β2 in such that α1 → α2

- **Dependency Preservation:** a decomposition is dependency preserving if at least one decomposed table satisfies every dependency. If a relation R is decomposed into relation R1 and R2, then the dependencies of R either must be a part of R1 or R2 or must be derivable from the combination of functional dependencies of R1 and R2.
  Example:
  - **R** (*A, B, C, D*)
    F = $\{A \to B, B \to C, C \to D, D \to A\}$
  - Decomposition: **R1**(A, B) **R2**(B, C) **R3**(C, D)
    - $A \to B$ is preserved on table R1
    - $B \to C$ is preserved on table R2
    - $C \to D$ is preserved on table R3
    - We have to check whether the one remaining FD: **D**→**A** is preserved or not.

R1	R2	R3
F_1={**A** → *AB*, **B** → *BA*}	F_2={**B** → *BC*, **C** → *CB*}	F_3={**C** → *CD*, **D** → *DC*}

    - $F' = F_1 \cup F_2 \cup F_3$.
    - Checking for: **D** → *A* in F'^{+}
      - *D* → *C* (from R3), *C* → *B* (from R2), *B* → *A* (from R1) : **D**→ **A** (By Transitivity)
        Hence all dependencies are preserved.

- **Prime Attributes:** attribute set that belongs to any candidate key
  Non-Prime Attributes: attribute set that does not belong to any candidate key

Normalisation:

- Normalization or Schema Refinement is a technique of organizing the data in the database to systematically decompose tables to eliminate data redundancy and undesirable characteristics
- Mainly used to reduce redundancy and to ensure data dependencies make sense
- Decompositions should be lossless and dependency preserving
- Informally, a relational database relation is often described as “normalized” if it meets third normal form. Most 3NF relations are free of insertion, update, and deletion anomalies

Normal Forms:

First Normal Form (1NF):

- The domains of all attributes of R are atomic (indivisible, multivalued/ composite attributes are non-atomic)
- The value of each attribute contains only a single value from that domain
- In DBMS, we assume that all the relations are in 1NF, by default
- Example of atomicity: Strings would normally be considered indivisible but in case of roll no. in the form CS0012 or EE1127 where the first two characters are extracted to find the department, the domain of roll numbers is not atomic
- Example:

Customer

Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025, 192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53; 182-929-2929
789	John	Doe	555-808-9633

Not in 1NF (telephone number is multivalued)

Customer Name

Customer ID	First Name	Surname
123	Pooja	Singh
456	San	Zhang
789	John	Doe

Customer Telephone Number

Customer ID	Telephone Number
123	555-861-2025
123	192-122-1111
456	(555) 403-1659 Ext. 53
456	182-929-2929
789	555-808-9633

In 1NF

Second Normal Form (2NF):

- Should be in 1NF
- Should **not** contain any **Partial Dependency**
- **Partial Dependency:** if a prime attribute derives a non-prime attribute. If a non-prime attribute(s) depends on a part of a candidate key (prime attributes)
- Example:

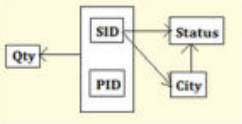
- **Supplier(SID, Status, City, PID, Qty)**

Supplier:

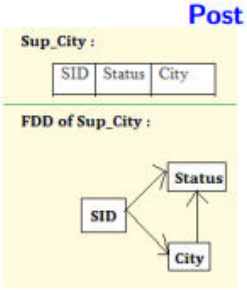
SID	Status	City	PID	Qty
S1	30	Delhi	P1	100
S1	30	Delhi	P2	125
S1	30	Delhi	P3	200
S1	30	Delhi	P4	130
S2	10	Karnal	P1	115
S2	10	Karnal	P2	250
S3	40	Rohtak	P1	245
S4	30	Delhi	P4	300
S4	30	Delhi	P5	315

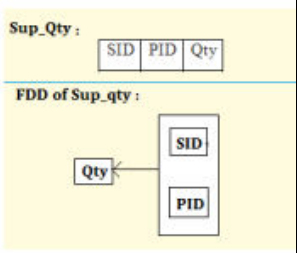
Key : (SID, PID)

Partial Dependencies:
$SID \rightarrow Status$
$SID \rightarrow City$



Post Normalization





Drawbacks:

- **Deletion Anomaly**: If we delete a tuple in *Sup_City*, then we not only loose the information about a supplier, but also loose the status value of a particular city.
- **Insertion Anomaly**: We cannot insert a City and its status until a supplier supplies at least one part.
- **Update Anomaly**: If the status value for a city is changed, then we will face the problem of searching every tuple for that city.

Third Normal Form (3NF):

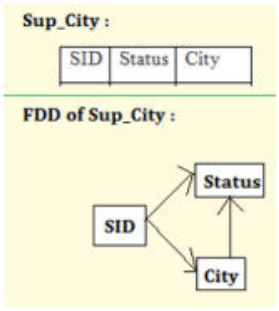
- Should be in 2NF
- Should **not** contain any **Transitive Dependency**
- **Transitive Dependency:** if a non-prime attribute derives a non-prime attribute. If an indirect relationship causes a functional dependency like A → B and B → C are true, then A → C happens to be a transitive dependency
- There is always a lossless-join, dependency-preserving decomposition into 3NF which is why it's the most common normal form
- Proper subset of any candidate key is not allowed to functionally determine the non-prime attributes
- Example:

Sup_City(SID, Status, City) (already in 2NF)

Sup_City:

SID	Status	City
S1	30	Delhi
S2	10	Karnal
S3	40	Rohtak
S4	30	Delhi

SID: Primary Key



- Redundancy?
  - Status
- Anomaly?
  - Yes

Functional Dependencies:
SID → Status,
SID → City,
City→ Status
Transitive Dependency :
SID → Status
{As SID → City and City → Status}

Post Normalization

SC:

SID	City
S1	Delhi
S2	Karnal
S3	Rohtak
S4	Delhi

SID: Primary Key

CS:

City	Status
Delhi	30
Karnal	10
Rohtak	40

City: Primary Key

The above two relations SC and CS are
- Lossless Join
- 3NF
- Dependency Preserving

- Algorithm to decompose to 3NF:
  - Eliminate redundant FDs, resulting in a canonical cover Fc of F
  - Create a relation Ri = XY for each FD X → Y in Fc
  - If the key K of R does not occur in any relation Ri, create one more relation Ri = K

- Example of decomposition:

  Relation schema: (cust_banker_branch) = (customer_id, employee_id, branch_name, type)

  The functional dependencies for this relation schema are (will be given):
  a) customer_id, employee_id → branch_name, type
  b) employee_id → branch_name
  c) customer_id, branch_name → employee_id

  We first compute a canonical cover
  Branch name is extraneous in the RHS of dependency (a) because the dependency is already included in dependency (b)

  No other attribute is extraneous, so we get Fc =
  a) customer_id, employee_id → type

b) employee_id → branch_name
c) customer_id, branch_name → employee_id

Creating a relation for each FD in Fc:

3NF schema: (customer_id, employee_id, type)
(employee_id, branch_name)
(customer_id, branch_name, employee_id)

(employee_id, branch_name) is a subset of the table (customer_id, branch_name, employee_id) so we remove that

The final simplified 3NF schema is:
(customer_id, employee_id, type)
(customer_id, branch_name, employee_id)

Boyce – Codd Normal Form (BCFN, 3.5NF):

- Should be in 3NF
- Every functional dependency (A → B), A is either the super key or the candidate key. In simple terms, for any case (A → B), A can't be a non-prime attribute. This is called **super key restriction**
- BCNF always eliminates redundancies/anomalies
- Either a lossless join is possible or preserving dependencies, both isn't possible

- Algorithm to decompose to BCNF:
  - For all dependencies A → B in F+, check if A is a superkey by using attribute closure
  - If not, then
    Choose a dependency in F + that breaks the BCNF rules, say A → B
    Create R1 = AB
    Create R2 = (R − (B − A))
  - Repeat for R1, and R2

- Example of decomposition:

  R = (A, B, C)
  F = {A → B, B → C}
  Key = {A}
  R is not in BCNF (B → C but B is not superkey)
  Decomposition:
  - R1 = (B, C)
  - R2 = (A, B)

Comparison of Normal Forms:

	1NF	2NF	3NF	BCNF
Properties	atomic attributes	should be in 1NF no partial dependency	should be in 2 NF no transitive dependency	should be in 3NF superkey restriction
Redundancy	high	high	high as compared to BCNF	none
Anomalies	may allow some	may allow some	may allow some	always eliminates
Composite Primary Key	allowed	allowed if no partial dependency	allowed	not allowed
Elimination	eliminate repeating groups	eliminate redundant data	eliminate coloumns not dependent on key	eliminate multiple candidate keys
Dependency preservation			preserves all the dependencies	may not preserve all dependencies
Summary	it is about shape of a record type	it is about the relationship between key and non-key fields	it is about the relationship between key and non-key fields	determinant should be a superkey

Multivalued Dependency:

- Multivalued dependency occurs when two attributes in a table are independent of each other but, both depend on a third attribute
- A multivalued dependency consists of at least two attributes that are dependent on a third attribute that's why it always requires at least three attributes
- For a dependency A → B, if for a single value of A, multiple values of B exist, then the relation will be a multi-valued dependency

- Example:

BIKE_MODEL	MANUF_YEAR	COLOR
M2011	2008	White
M2001	2008	Black
M3001	2013	White
M3001	2013	Black
M4006	2017	White
M4006	2017	Black

Here columns COLOR and MANUF_YEAR are dependent on BIKE_MODEL and independent of each other
BIKE_MODEL →→ MANUF_YEAR
BIKE_MODEL →→ COLOR
BIKE_MODEL multidetermined MANUF_YEAR and COLOR

- We use multivalued dependencies in two ways:
    - To test relations to determine whether they are legal under a given set of functional and multivalued dependencies
    - To specify constraints on the set of legal relations. We shall thus concern ourselves only with relations that satisfy a given set of functional and multivalued dependencies
- MVD Theory:

	Name	Rule
C-	Complementation	If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow (R - (X \cup Y))$.
A-	Augmentation	If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$.
T-	Transitivity	If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$.
	Replication	If $X \rightarrow Y$, then $X \twoheadrightarrow Y$ but the reverse is not true.
	Coalescence	If $X \twoheadrightarrow Y$ and there is a W such that $W \cap Y$ is empty, $W \rightarrow Z$ and $Y \supseteq Z$, then $X \rightarrow Z$.

- **Trivial MVD:** A →→ B in R is trivial; if B ⊆ A or; if A ∪ B = R
- **Closure set D+** is the set of all functional and multivalued dependencies logically implied by D

Fourth Normal Form (4NF):

- Should be in BCNF
- Has **no multivalued dependency**
- A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D + of the form A →→ B, where A ⊆ R and B ⊆ R, at least one of the following hold:
    - A →→ B is trivial (that is, β ⊆ α or α ∪ β = R)
    - A is a superkey for schema R

- Example:

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given student relation is in BCNF but not in 4NF because STU_ID multidetermined COURSE and HOBBY, so to make the above table into 4NF, we can decompose it into two tables:

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

- Algorithm to decompose to 4NF:
  - For all dependencies A →→ B in D +, check if A is a superkey by using attribute closure
  - If not, then
    Choose a dependency in F+ that breaks the 4NF rules, say A →→ B
    Create R1 = AB
    Create R2 = (R – (B – A))
  - Repeat for R1, and R2

- Example of decomposition:

  - R =(A, B, C, G, H, I)
    F = A ↠ B
    B ↠ HI
    CG ↠ H
  - R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R
  - Decomposition
    a) R_1 = (A, B) (R_1 is in 4NF)
    b) R_2 = (A, C, G, H, I) (R_2 is not in 4NF, decompose into R_3 and R_4)
    c) R_3 = (C, G, H) (R_3 is in 4NF)
    d) R_4 = (A, C, G, I) (R_4 is not in 4NF, decompose into R_5 and R_6)
    - $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI \rightarrow A \twoheadrightarrow HI$, (MVD transitivity), and
    - and hence A ↠ I (MVD restriction to R_4)

    e) R_5= (A, I) (R_5 is in 4NF)
    f) R_6 = (A, C, G) (R_6 is in 4NF)

- LIS Example for 4NF:

book_title	author_fname	author_lname	edition
DBMS CONCEPTS	BRINDA	RAY	1
DBMS CONCEPTS	AJAY	SHARMA	1
DBMS CONCEPTS	BRINDA	RAY	2
DBMS CONCEPTS	AJAY	SHARMA	2
JAVA PROGRAMMING	ANITHA	RAJ	5
JAVA PROGRAMMING	RIYA	MISRA	5
JAVA PROGRAMMING	ADITI	PANDEY	5
JAVA PROGRAMMING	ANITHA	RAJ	6
JAVA PROGRAMMING	RIYA	MISRA	6
JAVA PROGRAMMING	ADITI	PANDEY	6

Since the relation has no FDs, it is already in BCNF.
However, the relation has two nontrivial MVDs book title →→ {author fname, author lname} and book title →→ edition. Thus, it is not in 4NF.
Nontrivial MVDs must be decomposed to convert it into a set of relations in 4NF

Can be decomposed into the following relations:

book_title	author_fname	author_lname
DBMS CONCEPTS	BRINDA	RAY
DBMS CONCEPTS	AJAY	SHARMA
JAVA PROGRAMMING	ANITHA	RAJ
JAVA PROGRAMMING	RIYA	MISRA
JAVA PROGRAMMING	ADITI	PANDEY

book_title	edition
DBMS CONCEPTS	1
DBMS CONCEPTS	2
JAVA PROGRAMMING	5
JAVA PROGRAMMING	6

More Normal Forms:

- Elementary Key Normal Form (EKNF)
- Essential Tuple Normal Form (ETNF)
- Join Dependencies and Fifth Normal Form (5 NF)
- Sixth Normal Form (6NF)
- Domain/Key Normal Form (DKNF)

Temporal Databases

- Historical data: data collected about past events and circumstances pertaining to a particular subject, include time-dependent/time-varying data
- Examples: medical records, judicial records, share prices, exchange rates, interest rates, company profits
- **Temporal data:** have an associated time interval during which the data are valid
- **Temporal databases:** provide a uniform and systematic way of dealing with historical data
- **Snapshot:** the value of the data at a particular point in time

Temporal Database Theory:

- There are two different aspects of time in temporal databases:
  - **Valid Time:** time period during which a fact is true in real world, provided to the system
  - **Transaction Time:** time period during which a fact is stored in the database, based on transaction serialization order and is the timestamp generated automatically by the system

- Temporal Relation is one where each tuple has associated time; either valid time or transaction time or both associated with it:
  - **Uni-Temporal Relations:** has one axis of time, either Valid Time or Transaction Time
  - **Bi-Temporal Relations:** has both axis of time – Valid time and Transaction time. It includes Valid Start Time, Valid End Time, Transaction Start Time, Transaction End Time

- **Example:**

- John's Data In Non-Temporal Database

Date	Real world event	Address
April 3, 1992	John is born	
April 6, 1992	John's father registered his birth	Chennai
June 21, 2015	John gets a job	Chennai
Jan 10, 2016	John registers his new address	Mumbai

In a non-temporal database, John's address is entered as Chennai from 1992. When he registers his new address in 2016, the database gets updated and the address field now shows his Mumbai address. The previous Chennai address details will not be available. So, it will be difficult to find out exactly when he was living in Chennai and when he moved to Mumbai.

- John was born on April 3, 1992 in Chennai.
- His father registered his birth after three days on April 6, 1992.
- John did his entire schooling and college in Chennai.
- He got a job in Mumbai and shifted to Mumbai on June 21, 2015.
- He registered his change of address only on Jan 10, 2016.

- ## Uni-Temporal Relation (Adding Valid Time To John's Data)

Name	City	Valid From	Valid Till
John	Chennai	April 3, 1992	June 20, 2015
John	Mumbai	June 21, 2015	∞

- John was born on April 3, 1992 in Chennai.
- His father registered his birth after three days on April 6, 1992.
- John did his entire schooling and college in Chennai.
- He got a job in Mumbai and shifted to Mumbai on June 21, 2015.
- He registered his change of address only on Jan 10, 2016.

- The valid time temporal database contents look like this:
  `Name, City, Valid From, Valid Till`
- Johns father registers his birth on 6th April 1992, a new database entry is made:
  `Person(John, Chennai, 3-Apr-1992, ∞).`
- On January 10, 2016 John reports his new address in Mumbai:
  `Person(John, Mumbai, 21-June-2015, ∞).`
  - The original entry is updated:
    `Person(John, Chennai, 3-Apr-1992, 20-June-2015).`

- ## Bi-Temporal Relation (John's Data Using Both Valid And Transaction Time)

Name	City	Valid From	Valid Till	Entered	Superseded
John	Chennai	April 3, 1992	June 20, 2015	April 6, 1992	Jan 10, 2016
John	Mumbai	June 21, 2015	∞	Jan 10, 2016	∞

- John was born on April 3, 1992 in Chennai.
- His father registered his birth after three days on April 6, 1992.
- John did his entire schooling and college in Chennai.
- He got a job in Mumbai and shifted to Mumbai on June 21, 2015.
- He registered his change of address only on Jan 10, 2016.

- The database contents look like this:
  `Name, City, Valid From, Valid Till, Entered, Superseded`
- Johns father registers his birth on 6th April 1992:
  `Person(John, Chennai, 3-Apr-1992, ∞, 6-Apr-1992, ∞).`
- On January 10, 2016 John reports his new address in Mumbai:
  `Person(John, Mumbai, 21-June-2015, ∞, 10-Jan-2016, ∞).`
  - The original entry is updated as:
    `Person(John, Chennai, 3-Apr-1992, 20-June-2015, 6-Apr-1992 , 10-Jan-2016).`

- **Advantages:** the main advantages of this bi-temporal relations is that it provides historical (valid time) and roll back information (transaction time). For example, you can get the result for a query on John's history, like: Where did John live in the year 2001? The result for this query can be got with the valid time entry. The transaction time entry is important to get the rollback information

  Disadvantages: more storage, complex query processing, complex maintenance including backup and recovery

Application Programs and Architecture

Examples of Internet/Web or Mobile Application Programs:

- **Financial**:
  - Netbanking: SBI, PNB, BoB, Canara, HDFC, ICICI
  - Share Market: ICICIDirect, Sharekhan, HDFCDirect
  - Insurance & Investment: LICI, PolicyBazaar, NSDL, NPS,
  - Payment Gateway: Paytm, GPay, Bhim UPI, PhonePe,
  - e-Commerce: Amazon, Flipkart, eBay, BigBazaar, BigBasket,
- **Travel & Tourism**:
  - Travel Reservations: IRCTC, Airlines, MakeMyTrip, Yatra,
  - Accommodation: Booking, OYO, AirBnb, Fabhotels, Treebo,
  - Transportation: Uber, Ola Cab, Mega Cab, Meru Cab,
  - Navigation: Google Maps, MapQuest, Apple Maps,
  - Food & Delivery: Zomato, Swiggy, UberEats, Dunzo,
- **Communication**:
  - Live Interaction: Zoom, Google Meet, Teams, Webex, Skype,
  - Intermittent Interaction: WhatsApp, Telegram, Signal, Skype
  - Mail: Gmail, Yahoo, Hotmail, Rediffmail, Enterprise Mail,
  - Social Media: Facebook, Instagram, Twitter, YouTube,
- **Knowledge Discovery**:
  - Static: Google, Yahoo, Bing, Wikipedia, Encyclopedia.com,
  - Q&A: Quora, ASKfm, Yahoo Answers, Reddit, Digg,
- **Sports**:
  - Cricket: Cricbuzz, CricViz, Cricket-21, Cricket Exchange,
  - Tennis: ATP, ITF, SwingVision, TennisPAL, Tennis Clash,
- **Software Engineering**:
  - Issue Tracking: JIRA, BugZilla, Githubs, Gitlab,
  - VCS: Githubs, Gitlab, BitBucket, SourceForge,
  - Online IDE: OnlineGDB, Codechef, Ideone,
- **Library**:
  - Digital Library: National Digital Library of India,
  - Archives: Internet Archive, arXiv, Nextpoint,
- **Education**:
  - eLearning: BYJU's, IGNOU, NIIT, Edukart,
  - MOOCs: SWAYAM, edX, Coursera, Udemy,
- **Document Processing**:
  - Editing: Overleaf, Google Docs, Spreadsheet
  - Website, Blog: Google Sites, WordPress, Webly,
- **Health**:
  - Telemedicine: MDLIVE, Doctor on Demand,
  - National: Aarogy Setu, CoWin, NACO App,
- **Organizational ERP**: (Intranet)
  - Institutions: Students, Faculty, Course
  - Hospital: Patient, Doctor, OPD, IPD, Pharmacy,
  - Manufacturing: Suppliers, Inventory, Customers,
  - Bank: Customers, Accounts, Locker, Deposits,
  - Courier: Customers, Parcels, Delivery Agents,

- Applications are used to manage internal systems of big organisations (ERP: Enterprise Resource Planning)
- **Diversity** and **Unity** are characteristics of application programs

Application Functionality:

- **Frontend or Presentation Layer / Tier:** interacts with the user like display / view, input / output; responsible for providing the graphical user interface (GUI)
  Example: choose item, add to cart, checkout, pay, track order
  Interfaces may be browser-based, mobile app, or custom

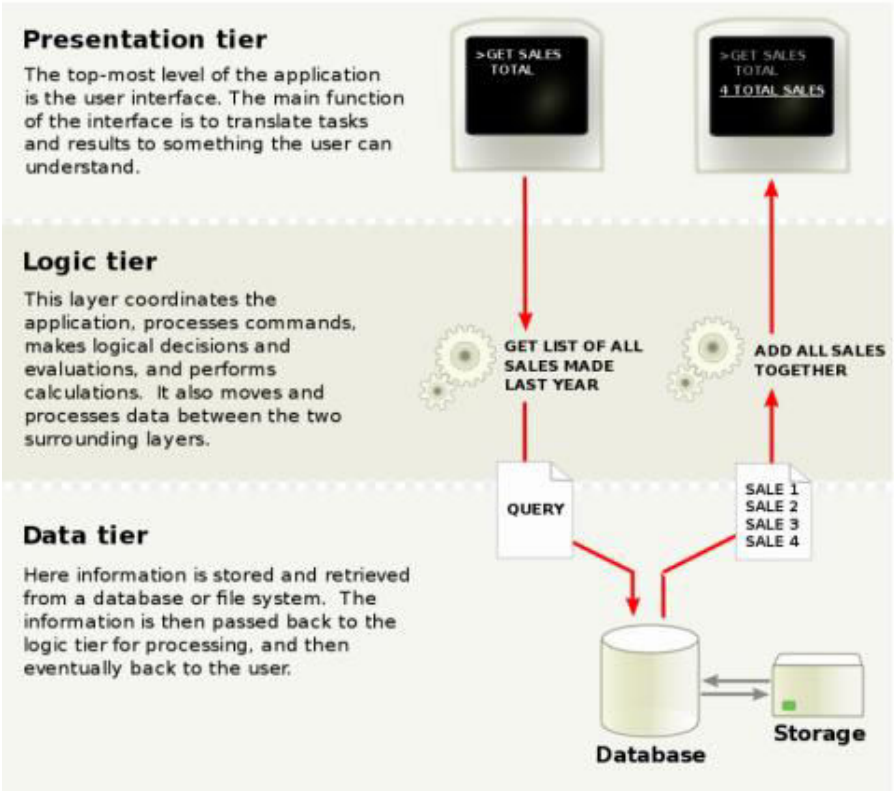
- **Middle or Application / Business Logic Layer / Tier**: implements the functionality of the application: links front and backend, differs based on the business logic, handles work flows, Support functionality based on frontend interface.
  Example: authentication, search / browse logic, pricing, cart management, payment handling (gateway), order management (mail / SMS / internal actions), delivery management
  Support functionality based on frontend interface

- **Backend or Data Access Layer / Tier**: manages persistent data, large volume, efficient access, security
  Example: User, Cart, Inventory, Order, Vendor databases

- All applications have this structural requirement which is translated into the following architecture

Application Architecture:

- Database architecture focuses on the design, development, implementation and maintenance of computer programs that store and organize information for businesses, agencies and institutions



- **Presentation Layer / Tier:**
  Model-View-Controller (MVC) architecture:
  Model: part of business logic that needs to be presented in the front end, example: data constraint checks, authentication etc
  View: presentation of data, depends on display device
  Controller: receives events, executes actions, and returns a view to the user

- **Business Logic Layer / Tier:** returns with a response for the Controller; provides high level view of data and actions on data, often using an object data model, hides details of data storage schema.
  Provides abstractions of entities (create objects)
  Has very complex business logic that can't be implemented as part of the database.
  Enforces business rules for carrying out actions
  Supports workflows which define how a task involving multiple participants is to be carried out

- **Data Access Layer / Tier:** interfaces between business logic layer and the underlying database, provides mapping from object model of business layer to relational model of database. Relational database design

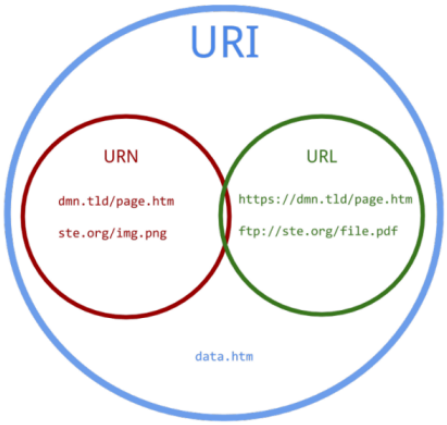
Architecture Classification:

- The design of a DBMS depends on its architecture. It can be **centralized**, **decentralized** or **hierarchical**

- The architecture of a DBMS can be seen as either single tier or multi-tier:
    - **1-tier architecture:** involves putting all of the required components for a software application or technology on a single server or platform, the simplest and most direct way
    - **2-tier architecture:** based on Client Server architecture,
    - **3-tier architecture:** Presentation, Logic and Data Access layers
    - **n-tier architecture:** an n-tier architecture distributes different components of the 3 tiers between different servers and adds interfaces tiers for interactions and workload balancing

Web Fundamentals:

- World Wide Web (**www**): The Web is a distributed information system based on hypertext
- **Hypertext**: text with references (hyperlinks) to other text that a reader can access
- HyperText Markup Language (**HTML**)
- Uniform Resource Identifier (**URI**): identifier of doc
- Uniform Resource Locator (**URL**): location of doc
- Uniform Resource Name (**URN**): name of doc
- HyperText Transfer Protocol (**HTTP**): used for communication with the Web server, connectionless
- **Session:** time for which a virtual connection to a web page remains
- **Cookie:** small piece of text containing identifying information, can be stored permanently or for a limited time
- **Web browser**: application software for accessing the www
- **Web server:** a software and underlying hardware that accepts requests via HTTP or its secure variant HTTPS
- Representation State Transfer (**REST**): allows use of standard HTTP request to a URL to execute a request and return data
- JavaScript Object Notation (**JSON**)



- **Script:** a list of (text) commands that are embedded in a web-page or in the server
- **Scripting language:** the programming languages in which scripts are written
- **Client-Side Scripting:** Client-side scripting is responsible for interaction within a web page. The client-side scripts are firstly downloaded at the client-end and then interpreted and executed by the browser; Javascript
- **Server-Side Scripting:** Server-side scripting is responsible for the completion or carrying out a task at the server-end and then sending the result to the client-end; Servlets, Java Server Pages (JSPs), PHP

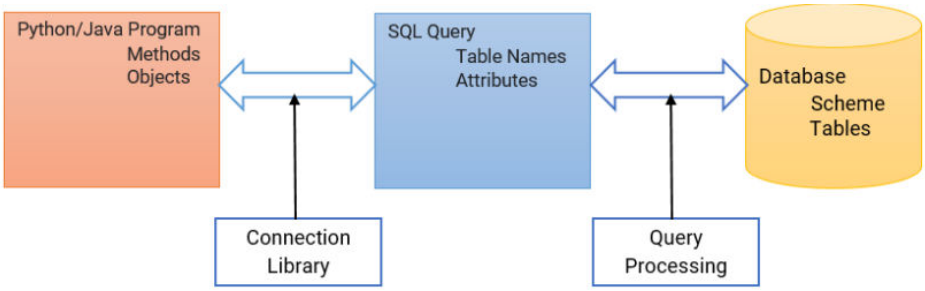
SQL and Native Language:

- **Application Programming Interface (API):** applications use it to interact with a database server
- Connect with database server → Send SQL commands to the database server → Fetch tuples of result one-by-one into program variables → close connection

- **Frameworks:**
  - **Connectionist:**
    Open Database Connectivity (ODBC): works with C, C++, C#, Visual Basic, and Python. Other data APIs include OLEDB, ADO.NET
    Java Database Connectivity (JDBC): works with Java

  - **Embedded SQL:** put SQL commands in some format inside native programming language, embedded SQL works with C, C++, Java, COBOL, FORTRAN and Pascal

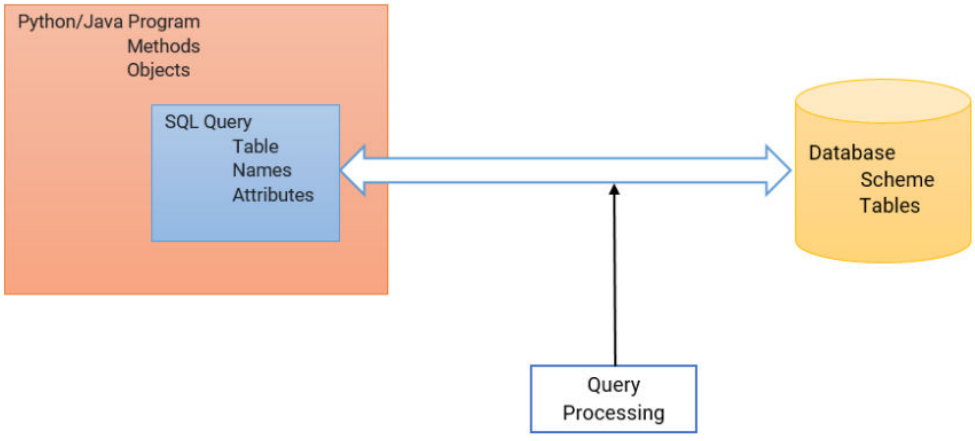
Connectionist Framework:



- **Open Database Connectivity (ODBC):**
  - is a standard API for accessing DBMS
  - aimed to be independent of database systems and operating systems
  - An application written using ODBC can be ported to other platforms, both on the client and server side, with few changes to the data access code
- **Java Database Connectivity (JDBC):**
  - is an API for the programming language Java, which defines how a client may access a database
  - It is a Java-based data access technology used for Java database connectivity

- **Connectionist Bridge Configurations:** a bridge is a special kind of driver that uses another driver-based technology. This driver translates source function-calls into target function-calls. Programmers usually use such a bridge when they lack a source driver for some database but have access to a target driver

Embedded SQL Framework:



- The SQL standard defines embedding of SQL in a variety of programming languages such as C, C++, Java, FORTRAN, and PL/1
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise embedded SQL

Python and PostgreSQL (VERY IMPORTANT FOR OPE):

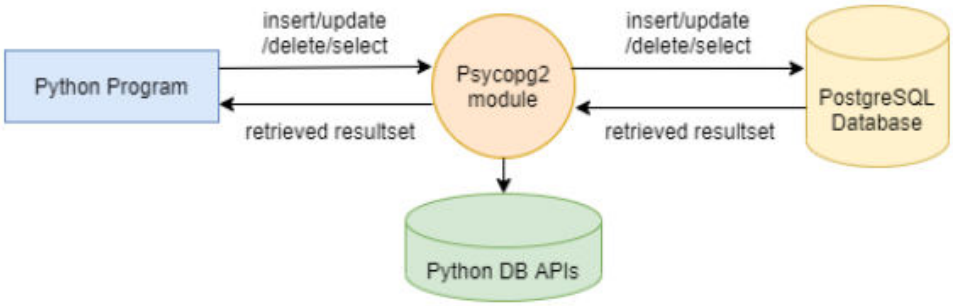
Install psycopg2: pip install psycopg2

Or

import psycopg2 (we will use this)

Steps to access PostgresSQL from Python using Psycopg

a) Create connection
b) Create cursor
c) Execute the query
d) Commit/rollback
e) Close the cursor
f) Close the connection



a) Create connection:
psycopg2.connect(database="dbname", user="username", password="mypass" host="127.0.0.1", port="5432")

b) Create cursor:
connection.cursor()

c) Execute the query:

cursor.execute(sql [, optional parameters])	execute sql query
cursor.executemany(sql, seq of parameters)	execute many sql queries
cursor.callproc(procname[, parameters])	call stored procedure
cursor.rowcount	return affected rows by previous execute
cursor.fetchone()	return the next row of a query result set
cursor.fetchmany([size=cursor.arraysize])	return many rows

d) Commit/ rollback (doesn't need to be used for SELECT):

connection.commit()	commit to transaction
connection.rollback()	roll back changes since last commit()

e) Close the cursor:
cursor.close()

f) Close the Connection:
connection.close()

- Examples:

  Connect to a PostgreSQL Database Server:

```
import psycopg2
def connectDb(dbname, usrname, pwd, address, portnum):
    conn = None
    try:
        # connect to the PostgreSQL database
        conn = psycopg2.connect(database = dbname, user = usrname, \
            password = pwd, host = address, port = portnum)
        print ("Database connected successfully")
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        conn.close()            # close the connection
connectDb("mydb", "myuser", "mypass", "127.0.0.1", "5432")  # function call
```

Output:

```
Database connected successfully
```

`psycopg2.DatabaseError`: Exception raised for errors that are related to the PostgreSQL database.
We assume the following for all the programs in this module:

- Database Name: *mydb*
- Username: *myuser*
- Password: *mypass*
- Host Name: *localhost* or IP address *127.0.0.1*

CREATE new PostgreSQL tables:

```
import psycopg2
def createTable():
    conn = None
    try:
        conn = psycopg2.connect(database = "mydb", user = "myuser", \
            password = "mypass", host = "127.0.0.1", port = "5432") # connect to the database
        cur = conn.cursor()  # create a new cursor
        cur.execute('''CREATE TABLE EMPLOYEE \
            (emp_num INT PRIMARY KEY     NOT NULL, \
            emp_name VARCHAR(40)    NOT NULL, \
            department VARCHAR(40)     NOT NULL)''') # execute the CREATE TABLE statement
        conn.commit()       # commit the changes to the database
        print ("Table created successfully")
        cur.close()         # close the cursor
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        if conn is not None:
            conn.close()    # close the connection
createTable()    #function call
```

Output (if table EMPLOYEE does not exist):

```
Table created successfully
```

Output (if table EMPLOYEE already exists):

```
relation "employee" already exists
```

Executing INSERT statement from Python:

```
import psycopg2
def insertRecord(num, name, dept):
    conn = None
    try:
        # connect to the PostgreSQL database
        conn = psycopg2.connect(database = "mydb", user = "myuser", \
            password = "mypass", host = "127.0.0.1", port = "5432")
        cur = conn.cursor()                 # create a new cursor
        # execute the INSERT statement
        cur.execute("INSERT INTO EMPLOYEE (emp_num, emp_name, department) \
            VALUES (%s, %s, %s)", (num, name, dept))
        conn.commit()                       # commit the changes to the database
        print ("Total number of rows inserted :", cur.rowcount);
        cur.close()                         # close the cursor
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        if conn is not None:
            conn.close()                    # close the connection
insertRecord(110, 'Bhaskar', 'HR')          #function call
```

Output:
```
Total number of rows inserted : 1
```

Output:
```
duplicate key value violates unique constraint "employee_pkey"
DETAIL:  Key (emp_num)=(110) already exists.
```

If a row already exists with emp_num = 110

Executing SELECT statement from Python:

```
import psycopg2
def selectAll():
    conn = None
    try:
        # connect to the PostgreSQL database
        conn = psycopg2.connect(database = "mydb", user = "myuser", \
            password = "mypass", host = "127.0.0.1", port = "5432")
        cur = conn.cursor()     # create a new cursor
        # execute the SELECT statement
        cur.execute("SELECT emp_num, emp_name, department FROM EMPLOYEE")
        rows = cur.fetchall()   # fetches all rows of the query result set
        for row in rows:
            print (print ("Employee ID = ", row[0], ", NAME = ", \
                row[1], ", DEPARTMENT = ", row[2]))
        cur.close()             # close the cursor
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        conn.close()            # close the connection
selectAll()                     # function call
```

Output:
```
Employee ID = 110, NAME = Bhaskar, DEPARTMENT = HR
Employee ID = 111, NAME = Ishaan, DEPARTMENT = FINANCE
Employee ID = 112, NAME = Jairaj, DEPARTMENT = TECHNOLOGY
Employee ID = 113, NAME = Ananya, DEPARTMENT = TECHNOLOGY
```

- Questions:

Instructions:

Note: Do not hard code the database name in your program, because your program will be run against a different database instance for evaluation.

For the database connection, use the following connection string variables:

database = `sys.argv[1]` //name of the database is obtained from the command line argument
user = os.environ.get('PGUSER')
password = os.environ.get('PGPASSWORD')
host = os.environ.get('PGHOST')
port = os.environ.get('PGPORT')

Problem Statement:

students	
student_fname	varchar(80)
student_lname	varchar(80)
roll_no	varchar(20)
department_code	varchar(4)
gender	varchar(1)
dob	date
degree	varchar(80)
mobile_no	numeric(10)

Write a Python program to print the roll number of the student. Student's first name is given in a file named 'name.txt' resides in the same folder as python program file.

The output of the python program is only roll number.

For example, if the first name of the student is 'Vikas'. Then output must be CS01 only. Note: No spaces.

Steps:

1. import relevant libraries (we need **sys**, **os** and **psycopg2**) [line 1]
2. store info about database, will be given in question[line 3 to 7]
3. open and read file content [line 9&10]
4. create connection [line 12]
5. create cursor [line 13]
6. execute query [line 15]
7. store results [line 16]
8. print results [line 18&19]
9. close cursor [line 21]
10. close connection [line 22]

```
1  import sys, os, psycopg2
2
3  database = sys.argv[1]
4  user = os.environ.get('PGUSER')
5  password = os.environ.get('PGPASSWORD')
6  host = os.environ.get('PGHOST')
7  port = os.environ.get('PGPORT')
8
9  file = open('name.txt', 'r')
10 fname = file.read()
11
12 conn = psycopg2.connect(database = database, user = user, password = password, host = host, port = port)
13 cur = conn.cursor()
14
15 cur.execute('select roll_no from students where student_fname = %s', (fname,))
16 rows = cur.fetchall()
17
18 for row in rows:
19     print(row[0])
20
21 cur.close()
22 conn.close()
```

Explanation:

- we need to import sys, os and psycopg2 because they help us connect to the database
- info about the database we need to connect to will be given in the question, store that info
- open file using command **open ('filename.txt', 'r'),** here filename is name.txt; and store in variable, here I used **file**
- read file using command **file.read()** and store in variable, here I used **fname** because the file contains the student's first name

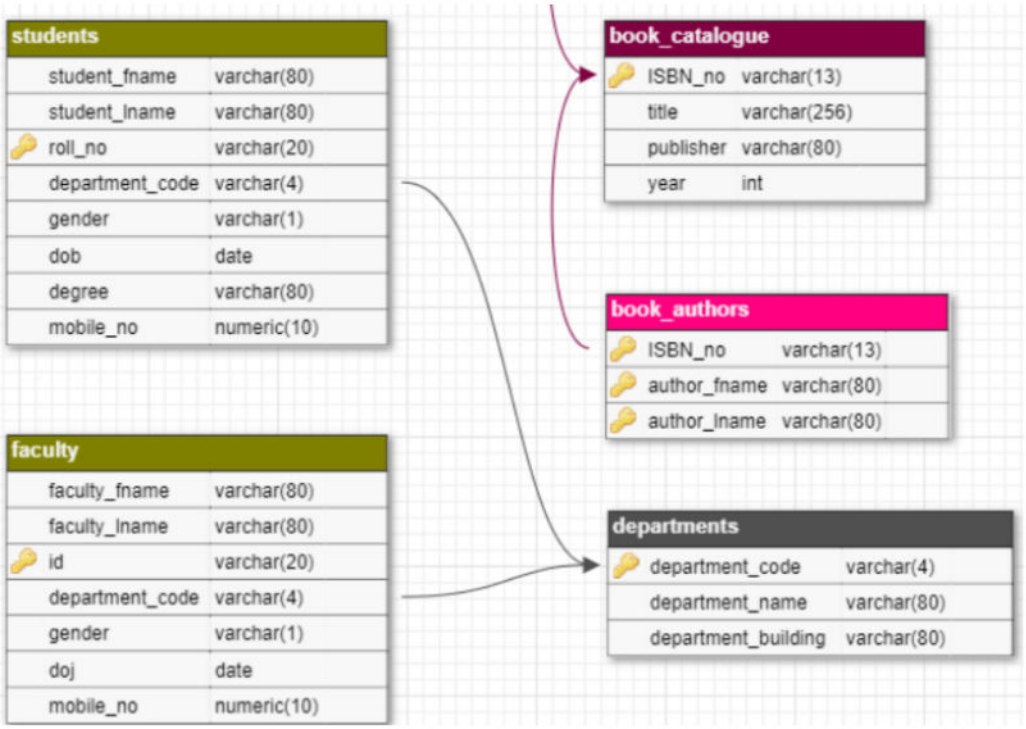
- after creating a connection and cursor, use cursor to execute select query
- question says to write a Python program to print the roll number of the student whose first name is given in a file named ‘name.txt’
- in the select query we select roll no from students where student_fname = %s [note: = %s is a placeholder and different from using string commands where we use like instead of =]
- the placeholder only accepts tuples so we have to mention the variable into which we read the file, the full format of the query will then be;
  (‘select roll no from students where student_fname = %s’ , (fname,))
- store the results from the query before printing them, the first part of the tuple contains the roll_no so we will only print that, hence **row[0]** is used while printing
- close the cursor and connection, it is considered good practice to do so

Instructions:

Note: Do not hard code the database name in your program, because your program will be run against a different database instance for evaluation.

For the database connection, use the following connection string variables:

```
database = sys.argv[1]      //name of the database is obtained from the command line argument
user = os.environ.get('PGUSER')
password = os.environ.get('PGPASSWORD')
host = os.environ.get('PGHOST')
port = os.environ.get('PGPORT')
```



Problem Statement:

Write a Python program to print the student's first name, the corresponding department name and the respective year of date of birth, if the year is even then print "Even" or else "Odd".

Student's first name is given in a file named 'name.txt' resides in the same folder as python program file.

The output of the python program is only student's first name, the corresponding department name and year of date of birth, if the year is even then "Even" or else "Odd".

For example, 'Suman' and 'Computer Science' is the name and department name of the student. '2002' is the year he was born in. '2002' is even. Then, the final output will be **Suman,Computer Science,Even** only. Note: No spaces.

For example, 'Vinod' and 'Electrical Engineering' is the name and department name of the student. '2003' is the year he was born in. '2003' is not even. Then, the final output will be **Vinod,Electrical Engineering,Odd** only. Note: No spaces.

```
1  import os, sys, psycopg2, datetime
2
3  database = sys.argv[1]
4  user = os.environ.get('PGUSER')
5  password = os.environ.get('PGPASSWORD')
6  host = os.environ.get('PGHOST')
7  port = os.environ.get('PGPORT')
8
9  file = open('name.txt', 'r')
10 student_fname = file.read()
11
12 conn = psycopg2.connect(database = database, user = user, password = password, host = host, port = port)
13 cur = conn.cursor()
14
15 cur.execute('select s.student_fname, d.department_name, s.dob from students s, departments d where s.department_code = d.department_code and s.student_fname = %s', (student_fname,))
16 rows = cur.fetchall()
17
18 for row in rows:
19     if (row[2].year % 2 == 0):
20         print(row[0] + ',' + row[1] + ',' + 'Even')
21     else:
22         print(row[0] + ',' + row[1] + ',' + 'Odd')
23
24
25 cur.close()
26 conn.close()
27
```

Steps:

1. import relevant libraries (we need **sys**, **os**, **psycopg2** and **datetime**) [line 1]
2. store info about database, will be given in question [line 3 to 7]
3. open and read file content [line 9&10]
4. create connection [line 12]
5. create cursor [line 13]
6. execute query [line 15]
7. store results [line 16]
8. print results [line 18 to 22]
9. close cursor [line 25]
10. close connection [line 26]

Explanation:

- we need to import **datetime** because the question requires us to extract the year from date of birth which is a datetime object
- after the results have been stored, proceed to extract the year and check if its odd or even
- row[2] stores the dob (row[0] stores name and row[1] stores department name), use function row[2].year to extract the year
- format the string that needs to be printed using +

<u>Web and Internet Development using Python:</u>

Python offers several frameworks such as **bottle.py**, **Flask**, **CherryPy**, **Pyramid**, **Django** and **web2py** for web development.

- **Python offers many choices for web development**
  - Frameworks such as **Django** and **Pyramid**.
  - Micro-frameworks such as **Flask** and **Bottle**.
  - Advanced content management systems such as **Plone** and **django CMS**.
- **Python's standard library supports many internet protocols**
  - **HTML** and **XML**
  - **JSON**
  - **E-mail** processing
  - Support for **FTP**, **IMAP**, and other Internet protocols
  - Easy-to-use socket interface
- **The package Index has more libraries**
  - **Requests**, a powerful HTTP client library.
  - **Beautiful Soup**, an HTML parser that can handle all sorts of HTML.
  - **Feedparser** for parsing RSS/Atom feeds.
  - **Paramiko**, implementing the SSH2 protocol.
  - **Twisted Python**, a framework for asynchronous network programming.

Rapid Application Development (RAD):

- RAD Software is an agile model that focuses on fast prototyping and quick feedback in app development to ensure speedier delivery and an efficient result
- **Platforms and Tools**: G Suite, Google App Engine, Microsoft Azure, Amazon Elastic Compute Cloud (EC2), AWS Elastic Beanstalk
- Web application development **frameworks**: Java Server Faces (JSF), Ruby on Rails

Algorithms and Data Structures

	Data Structure	Time Complexity								Space Complexity
		Average				Worst				Worst
		Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Linear Data Structures	Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
	Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Non-Linear Data Structures	Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
	Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
	Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
	Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
	B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
	AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

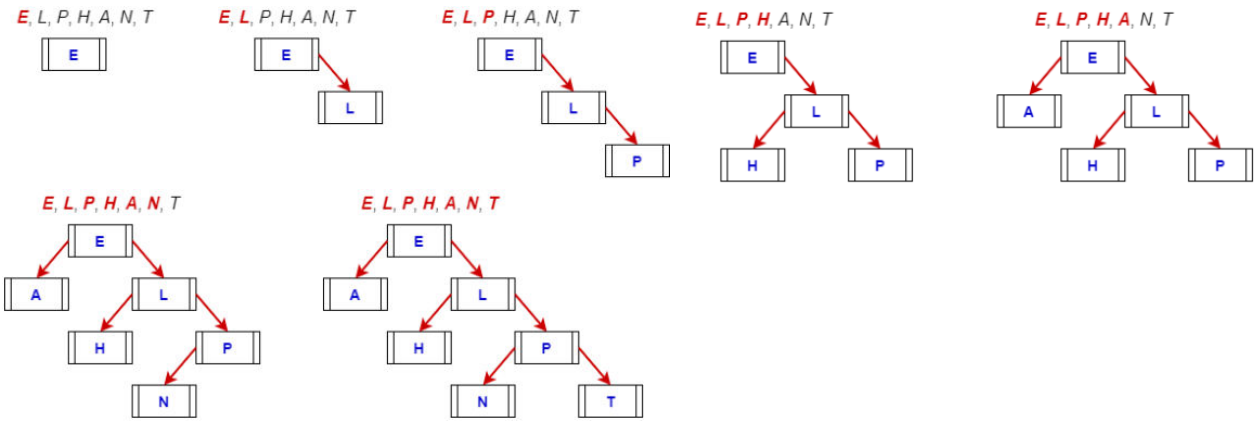
Linear vs Non-linear Data Structures:

- **Data structure**: A data structure specifies the way of organizing and storing in-memory data that enables efficient access and modification of the data
- **Linear data structures:** a Linear data structure has data elements arranged in linear or sequential manner such that each member element is connected to its previous and next element
    - Array: the data elements are stored at contiguous locations in memory
    - Linked List: the data elements are not required to be stored at contiguous locations in memory. Rather each element stores a link (a pointer to a reference) to the location of the next element
    - Queue: it is a FIFO (First in First Out) data structure
    - Stack: it is a LIFO (Last in First Out) data structure
- **Non-Linear data structures**: those data structures in which data items are not arranged in a sequence and each element may have multiple paths to connect to other elements
    - Graph: Undirected or Directed, Unweighted or Weighted, and variants
    - Tree: Rooted or Unrooted, Binary or n-ary, Balanced or Unbalanced, and variants
    - Hash Table: Array with lists (coalesced chains) and one or more hash functions
    - Skip List: Multi-layered interconnected linked lists

Linear Data Structure	Non-Linear Data Structure
• Data elements are *arranged* in a linear order where each and every elements are attached to its previous and next adjacent	• Data elements are *arranged* in hierarchical or networked manner
• Single *level* is involved	• Multiple *level* are involved
• *Implementation* is easy in comparison to non-linear data structure	• *Implementation* is complex in comparison to linear data structure
• Data elements can be *traversed* in one way only	• Data elements can be *traversed* in multiple ways. Various traversals may be defined to linearize the data: Depth-First, Breadth-First, Inorder, Prepoder, Postorder, etc.
• *Examples*: array, stack, queue, linked list, and their variants	• *Examples*: trees, graphs, skip list, hash map, and several variants

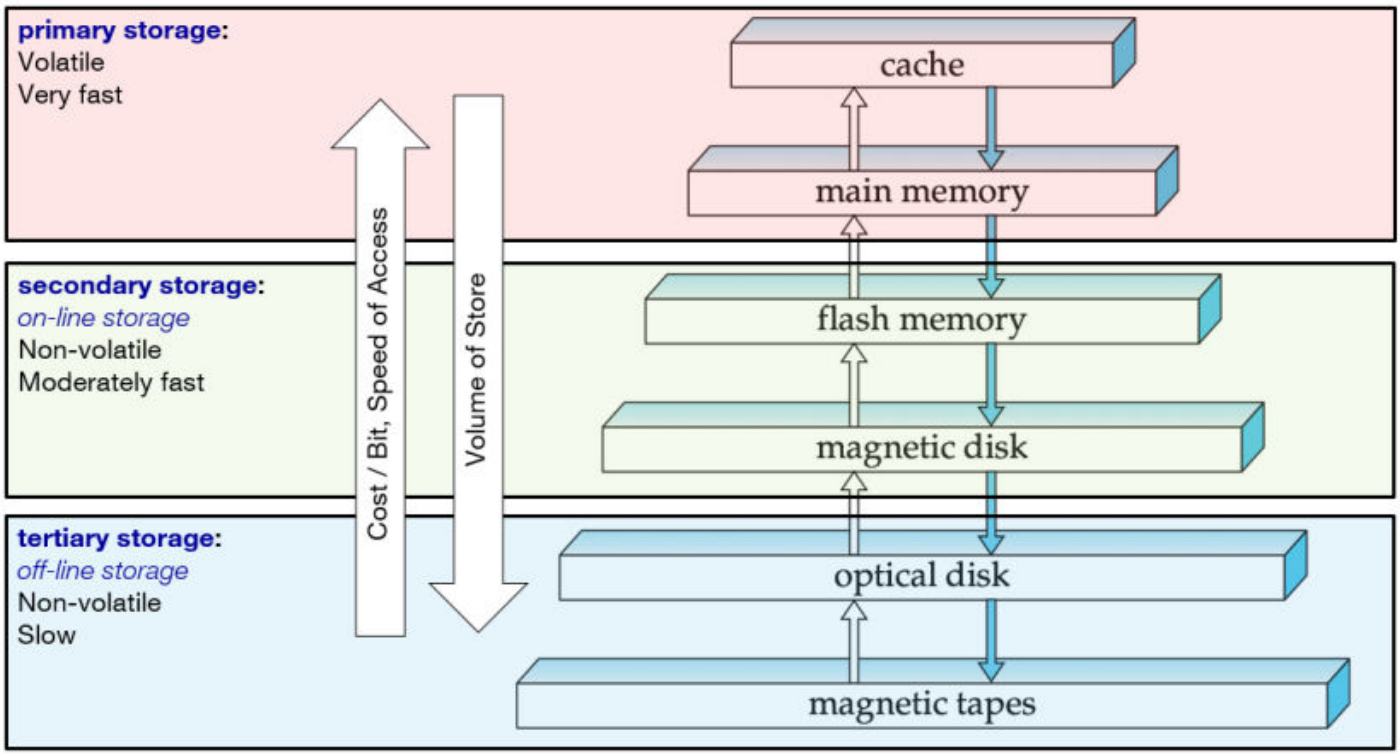
Binary Search Tree:

- A binary search tree is a binary tree that is either empty or satisfies the following conditions: For each node V in the Tree:
    - The value of the left child or left subtree is always less than the value of V
    - The value of the right child or right subtree is always greater than the value of V
- in a complete BST, the max no. of nodes at height h is **2^h - 1**
- Example: obtain the BST by inserting the following values- E, L, P, H, A, N, T



Storage and File Structure

- **Volatile storage:** loses contents when power is switched of
- **Non-Volatile storage:** contents persist even when power is switched off; includes secondary and tertiary storage, as well as battery-backed up main-memory

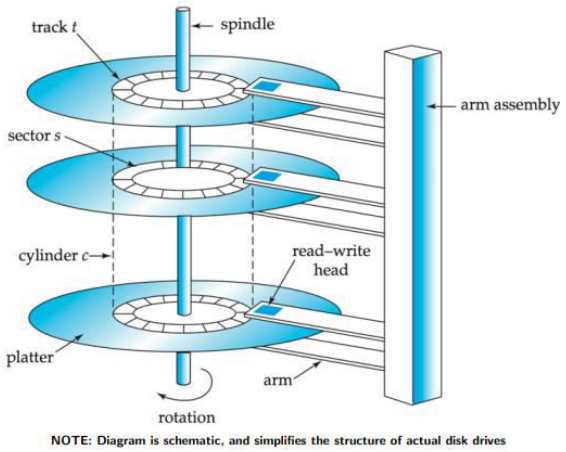


Primary Storage:

- **Cache:** fastest and most costly form of storage, volatile, managed by the computer system hardware
- **Main memory:** fast access, expensive, too small

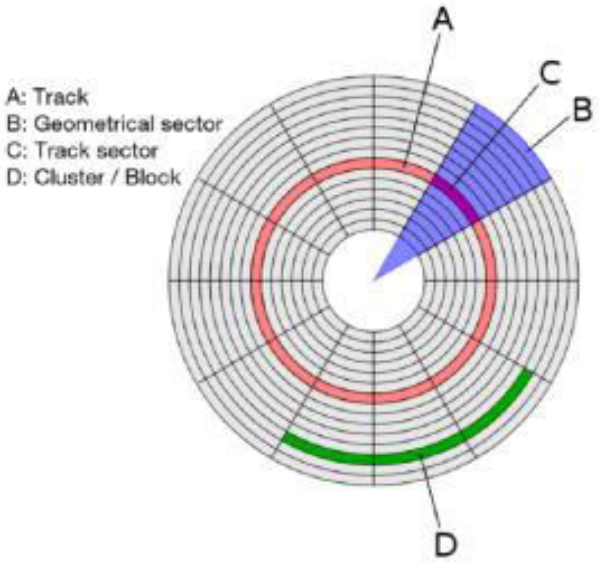
Secondary Storage:

- **Flash memory:** non-volatile, can support a limited number of write/erase cycles, reads are roughly as fast as main memory, writes are slow (few microseconds), erase is slower
- **Magnetic Disk (IMPORTANT):**
  - data is stored on spinning disk, and read/written magnetically
  - data must be moved from disk to main memory for access, and written back for storage - much slower access than main memory
  - direct-access
  - non-volatile
  - large capacity



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

- both surfaces can be read at the same time
- **tracks:** surface of platter divided into circular tracks
- **sectors:** each track is divided into sectors; a sector is the smallest unit of data read or written; sector size typically 512 bytes; all sectors are the same size
- The number of cylinders of a disk drive exactly equals the number of tracks on a single surface in the drive



❖ **Additional info:** to convert from one unit to another divide/multiply by 1024.
Bytes(b) < Kilobyte (KB) < Megabyte (MB) < Gigabyte (GB) < Terabyte (TB)

- **Disk Controller:** interfaces between the computer system and the disk drive hardware; accepts high-level commands to read or write a sector; initiates actions moving the disk arm to the right track, reading or writing the data; computes and attaches checksums to each sector to verify that correct read back; ensures successful writing by reading back sector after writing it; performs remapping of bad sectors
- **Access Time:** time from a read or write request issue to start of data transfer:
  Seek Time: time to reposition the arm over the correct track
  Rotational Latency: time for the sector to be accessed to appear under the head; Rotational latency = (1 / 2) x (1 / rotational speed) x 1000
  Total access time = seek time + rotational latency
- **Data-transfer Rate:** the rate at which data can be retrieved from or stored to the disk
- **Mean Time To Failure (MTTF):** avg. time the disk is expected to run continuously without any failure
- **Example questions:**

Question
Consider a magnetic disk with the following specifications. The magnetic disk consists of 16 platters, and has information recorded on both the surfaces of each platter. Each platter's surface is logically divided into 128 tracks, each of which is subdivided into 256 sectors. Find the storage capacity of a track. (Given, sector size is 512 bytes.)
Answer
To find the storage capacity of a track, we need to know the number of sectors in a track and the size of each sector.
We know that each platter has 128 tracks and there are 16 platters, so the total number of tracks is:
128 tracks/platter x 16 platters = 2048 tracks
We also know that each track is logically divided into 256 sectors and each sector has a size of 512 bytes. Therefore, the storage capacity of a track is:
256 sectors/track x 512 bytes/sector = 131,072 bytes/track
Therefore, the storage capacity of a track is 131,072 bytes or 128 kilobytes

Question:
Consider a magnetic disk with 8 platters, 2 surfaces/platter, 1024 tracks/surface, 2048 sectors/track, and 512 bytes/sector. The disk rotates with 6000 revolutions per minute and seek time is 3ms. What is the capacity of the disk? What will be the rotational latency? What is the minimum number of bits required for addressing all the sectors?

Answer:
Capacity –
To calculate the capacity of the disk, we first need to find the total number of sectors on the disk, and then multiply that by the size of each sector.
The total number of sectors on the disk can be calculated as follows:
Number of platters = 8 Number of surfaces/platters = 2 Number of tracks/surfaces = 1024 Number of sectors/tracks = 2048
Total number of sectors = Number of platters x Number of surfaces/platter x Number of tracks/surface x Number of sectors/track = 8 x 2 x 1024 x 2048 = 33,554,432 sectors
The size of each sector is given as 512 bytes. Therefore, the total capacity of the disk can be calculated as:
Capacity of disk = Total number of sectors x Size of each sector = 33,554,432 sectors x 512 bytes/sector = 17,179,869,184 bytes or 16GB

Rotational Latency-
Rotational latency = (1/2) x (1/rotational speed) x 1000
where rotational speed is the speed of the disk in revolutions per second.
In this case, the rotational speed is given as 6000 revolutions per minute, or 100 revolutions per second. Therefore, we can calculate the rotational latency as:
Rotational latency = (1/2) x (1/100) x 1000 = 5 milliseconds

Minimum number of bits-
In computer storage systems, each sector is assigned a unique identifier called a sector address. The sector address is used to locate the specific sector on the disk when data is read from or written to the disk.
To address each sector, we need a unique binary number that identifies the sector. The minimum number of bits required for addressing all the sectors can be calculated as follows:

Number of sectors = Number of platters x Number of surfaces/platter x Number of tracks/surface x Number of sectors/track = 8 x 2 x 1024 x 2048 = 33,554,432 sectors
To address 33,554,432 sectors, we need at least log2(33,554,432) bits.
log2(33,554,432) = 24.99999968
Therefore, we need 25 bits to address all the sectors
2^(25) = 33,554,432

Tertiary Storage:

- **Optical Disks:** non-volatile; data is read optically from a spinning disk using a laser
- **Magnetic Tapes:** hold large volumes of data and provide high transfer rates; tapes are cheap, but cost of drives is very high; very slow access time in comparison to magnetic and optical disks; limited to sequential access; used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another

Cloud Storage:

- Cloud storage is purchased from a third-party cloud vendor who owns and operates data storage capacity and delivers it over the Internet in a pay-as-you-go model
- These cloud storage vendors manage capacity, security and durability to make data accessible to applications all around the world
- Applications access cloud storage through traditional storage protocols or directly via an API

Other Storages:

- Flash Drives: USB flash drives are removable and rewritable storage devices that, as the name suggests, require a USB port for connection and utilizes non-volatile flash memory technology
- A Secure Digital (SD Card): a type of removable memory card used to read and write large quantities of data
- Flash Storage
- Solid-State Drives (SSD): SSDs replace traditional mechanical hard disks by using flash-based memory, which is significantly faster; SSDs speed up computers significantly due to their low read-access times and fast throughput

File Structure

File Organisation:

- A **database** is a collection of files; A **file** is a sequence of records; a **record** is a sequence of fields
- A database file is partitioned into fixed-length storage units called **blocks**

Organization of Records in Files:

- **Heap:** a 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
- **Hashing:** a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

- In a **multitable clustering file organization** record of several different relations can be stored in the same file

Data Dictionary Storage:

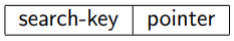
- Data Dictionary (also, System Catalog) stores metadata (data about data) such as:
  - Information about relations
  - User and accounting information, including passwords
  - Statistical and descriptive data
  - Physical file organization information
  - Information about indices

Storage Access:

- Database system seeks to minimize the number of block transfers between the disk and memory
- A database file is partitioned into fixed-length storage units called **blocks**
- **Buffer:** portion of main memory available to store copies of disk blocks
- **Buffer Manager:** subsystem responsible for allocating buffer space in main memory
- Most operating systems replace the block **least recently used (LRU strategy)**
- **Pinned block:** memory block that is not allowed to be written back to disk
- **Toss-immediate strategy**: frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy:** system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block
- Buffer managers also support **forced output** of blocks for the purpose of recovery

Indexing and Hashing

Basic Concepts:

- Indexing mechanisms are used to speed up access to desired data
- **Search Key:** attribute to set of attributes used to look up records in a file
- An **index file** consists of records (called **index entries**) of the form 

- Index files are typically much smaller than the original file
- **Ordered indices:** search keys are stored in sorted order, update is costlier
- **Hash indices:** search keys are distributed uniformly across buckets using a hash function
- Index Evaluation Metrics: access types supported efficiently, access time, insertion time, deletion time, space overhead

Ordered Indices:

- Index entries are stored sorted on the search key value
- **Primary index/ clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file; the search key of a primary index is usually but not necessarily the primary key

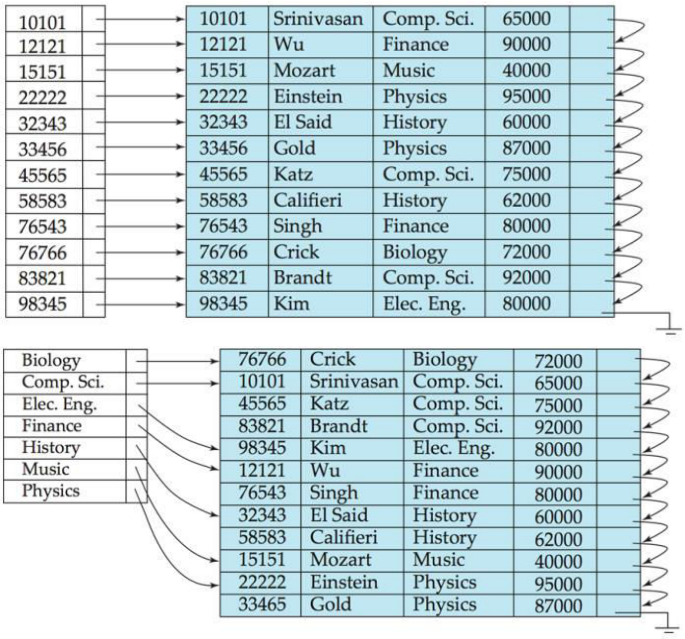
Secondary index/ non-clustering index: an index whose search key specifies an order different from the sequential order of the file, have to be dense

Index on "Name"	
Name	*Pointer*
Anupam Basu	2
Pabitra Mitra	6
Partha Pratim Das	1
Prabir Kumar Biswas	7
Rajib Mall	5
Ranjan Sen	3
Sudeshna Sarkar	4

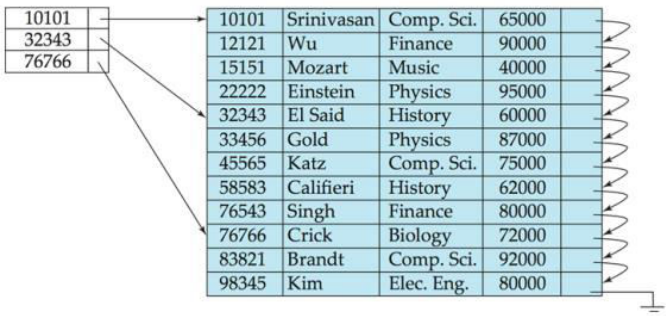
Table "Faculty"		
Rec #	*Name*	*Phone*
1	Partha Pratim Das	81998
2	Anupam Basu	82404
3	Ranjan Sen	84624
4	Sudeshna Sarkar	82432
5	Rajib Mall	83668
6	Pabitra Mitra	81664
7	Prabir Kumar Biswas	84772

Here, since name is the search key that the file is sorted by (blue table) it is the primary index and the phone number is the secondary index

- **Index-sequential file:** ordered sequential file with a primary index
- **Dense index:** index points to every search-key value in the file, very large in size; examples:



- **Sparse index:** contains index records for only some search-key values, less space and less maintenance overhead for insertions and deletions, generally slower than dense index for locating records; examples:



Balanced Binary Search Trees:

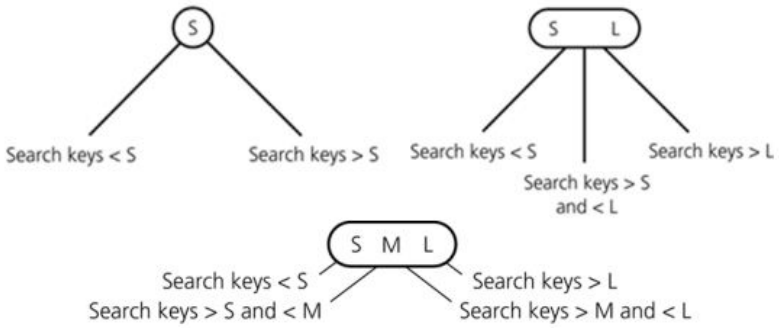
- Study PDSA lol
- Bad Tree: h ~ O(n)
- Good Tree: h ~ O(logn)

2-3-4 Tree:

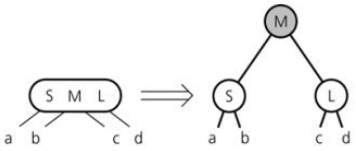
- All leaves are at the same depth (the bottom level)
- Height, h, of all leaf nodes are same.
- h ~ O(logn)
- Complexity of search, insert and delete: O(h) ~ O(logn)
- All data is kept in sorted order
- Every node (leaf or internal) is a 2-node, 3-node or a 4-node (based on the number of links or children), and holds one, two, or three data elements, respectively

Uses 3 kinds of nodes satisfying key relationships as shown below:

- A 2-node must contain a single data item (S) and two links
- A 3-node must contain two data items (S, L) and three links
- A 4-node must contain three data items (S, M, L) and four links
- A leaf may contain either one, two, or three data items



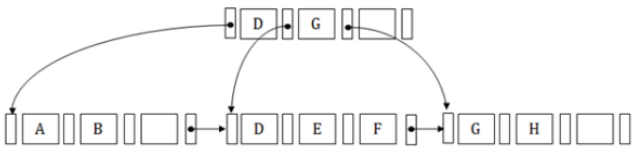
Splitting at Root



- Each node that is not a root or a leaf has between n/2 and n children
- A leaf node has between (n−1)/2 and n − 1 values

B+ Tree Index Files:

- Is a balanced binary search tree
- Follows a multi-level index format like 2-3-4 Tree
- Has the leaf nodes denoting actual data pointers
- Ensures that all leaf nodes remain at the same height (like 2-3-4 Tree)
- Has the leaf nodes are linked using a link list
- Can support random access as well as sequential access



Internal node contains

- At least $\frac{n}{2}$ child pointers, except the root node
- At most n pointers

Leaf node contains

- At least $\frac{n}{2}$ record pointers and $\frac{n}{2}$ key values
- At most n record pointer and n key values
- One block pointer P to point to next leaf node

Note: These are approximate values, we will discuss more precise values later in this lecture.

- p-1 is the max no. of keys in a node where p is the order; p is the max no. of children
- Data point doesn't need to be repeated in intermediate step

B-Tree:

- B-tree allows search-key values to appear only once
- 3 order B tree means that nodes can have 2 data points and three children
- In a B-tree of order m, a non-root node can have at least ⌈m/2⌉ child pointers and ⌈m/2⌉ - 1 keys

Questions:

1. A telecom company has 2^16 customer records in a table T in their database. These records are sorted in ascending order of the attribute customer_id, which is also the primary key of T. The data file is stored in a disk with a block size of 512 bytes. Assume that, in each block, the records are unspanned and are of fixed-length. Each record is of size 32 bytes, the size of the primary key field is 10 bytes and the size of the block pointer is 6 bytes. If a primary (sparse index with an index entry for every block in the file) index is created on the data file, what is the minimum number of blocks required for the index file?
   Options: A. 64
   B. 128
   C. 256
   D. 512

   Answer:

- Calculate the total size of the data file: Total size of data file = Number of customer records * Size of each record = 2^16 * 32 bytes = 2,097,152 bytes
- Calculate the number of blocks required to store the data file: Number of blocks for data file = Total size of data file / Block size = 2,097,152 bytes / 512 bytes = 4096 blocks
- Calculate the size of each index entry: Size of each index entry = Size of primary key field + Size of block pointer = 10 bytes + 6 bytes = 16 bytes
- Since primary index is a sparse index with an index entry for every block in the file, no. of entries is the same as total no. of blocks which is 4096
- Calculate the size of the index file: Size of index file = Number of index entries * Size of each index entry = 4096 entries * 16 bytes = 65,536 bytes
- Calculate the number of blocks required to store the index file: Number of blocks for index file = Size of index file / Block size = 65,536 bytes / 512 bytes = 128 blocks

Transactions

- A **transaction** is a unit of program execution that accesses and, possibly updates, various data items
- **Failures** of various kinds, such as hardware failures and system crashes and **concurrent execution** of multiple transactions are the two main issues to deal with in transactions

- **Example:**

Transaction to transfer $50 from account A to account B:

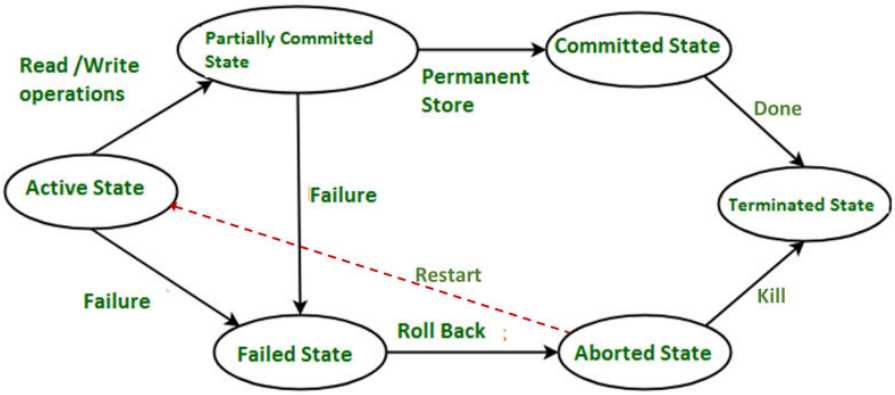
1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

Required Properties of a Transaction: **ACID** (Important):

- **Atomicity:** in the example, if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
  - Atomicity guarantees that all of the commands that make up a transaction are treated as a single unit and either succeed or fail together. So, the system should ensure that updates of a partially executed transaction are not reflected in the database
  - All or nothing
- **Consistency:** consistency guarantees that changes made within a transaction are consistent with database constraints. This includes all rules, constraints, and triggers. If the data gets into an illegal state, the whole transaction fails
  - Preserves database integrity
  - For example, there is a constraint that the balance should be a positive integer. If we try to overdraw money, then the balance won't meet the constraint. Because of that, the consistency of the ACID transaction will be violated and the transaction will fail
- **Isolation:** isolation ensures that all transactions run in an isolated environment. That enables running transactions concurrently because transactions don't interfere with each other
  - Execute as if they were run alone
- **Durability:** durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (like power outage or crash). Durability guarantees that once the transaction completes and changes are written to the database, they are persisted. This ensures that data within the system will persist even in the case of system failures like crashes or power outages
  - Results are not lost by a failure

Transaction States:

- **Active:** the initial state; the transaction stays in this state while it is executing
- **Partially committed:** after the final statement has been executed
- **Failed:** after the discovery that normal execution can no longer proceed
- **Aborted:** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted: restart the transaction (can be done only if no internal logical error) or kill the transaction
- **Committed:** after successful completion
- **Terminated:** after it has been committed or aborted (killed)



Concurrent Executions:

- Multiple transactions are allowed to run concurrently in the system; increased processor and disk utilization; reduced average response time
- **Concurrency Control Schemes:** mechanisms to achieve isolation
- **Schedule:** A sequence of instructions that specify the chronological order in which instructions of concurrent transactions are executed
    - Must consist of all instructions of those transactions
    - Must preserve the order in which the instructions appear in each individual transaction
    - **Schedule 1:**
        - Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B
        - An example of a **serial** schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
50	250	300	T1, write B	@ Commit
45	250	295	T2, write A	
45	255	300	T2, write B	@Commit

Consistent @ Commit
Inconsistent @ Transit
Inconsistent @ Commit

    - **Schedule 2:**

A **serial** schedule in which T_2 is followed by T_1 :

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
90	200	290	T2, write A	
90	210	300	T2, write B	@ Commit
40	210	250	T1, write A	
40	260	300	T1, write B	@Commit

Consistent @ Commit
Inconsistent @ Transit
Inconsistent @ Commit

Values of A & B are different from Schedule 1 – yet consistent

    - **Schedule 3:**
        - Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is **equivalent** to Schedule 1

Schedule 3

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 1

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
45	200	245	T2, write A	
45	250	295	T1, write B	@ Commit
45	255	300	T2, write B	@Commit

Consistent @ Commit
Inconsistent @ Transit
Inconsistent @ Commit

Note – In schedules 1, 2 and 3, the sum "$A + B$" is preserved

    - **Schedule 4:**

- The following concurrent schedule does not preserve the sum of "$A + B$"

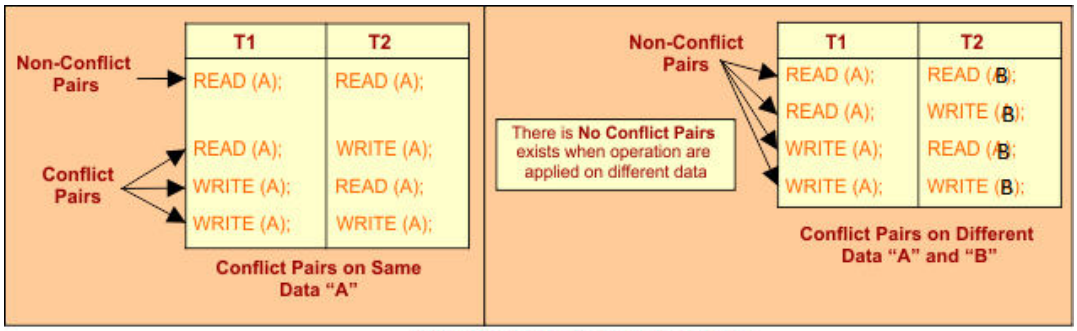
T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
90	200	290	T2, write A	
50	200	250	T1, write A	
50	250	300	T1, write B	@ Commit
50	210	260	T2, write B	@ Commit

Consistent @ Commit
Inconsistent @ Transit
Inconsistent @ Commit

Serializability:

- Serializability helps to ensure Isolation and Consistency of a schedule
- **Assumption:** Each transaction preserves database consistency
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule
- **Conflicting operations:** Read and Write are considered as conflicting operations, if they hold the following conditions:
  - Both the operations are on the same data
  - Both the operations (Read and Write) belong to different transactions
  - At least one of the two operations is a write operation
- **Non-conflicting operations:** Read and Write are considered as non-conflicting operations, if they hold the following conditions:
  - Both the operations are on different data item
  - Both the operations belong to different transactions



Conflict & Non-Conflict Paris in DBMS

Conflict Serializability:

- **Conflict equivalent:** if a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict equivalent
- **Conflict Serializability:** we say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule
- Example of a conflict serializable schedule:

Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions:

- Swap T1.read(B) and T2.write(A)
- Swap T1.read(B) and T2.read(A)
- Swap T1.write(B) and T2.write(A)
- Swap T1.write(B) and T2.read(A)

These swaps do not conflict as they work with different items (A or B) in different transactions

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

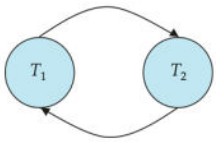
- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

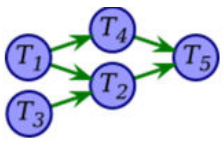
We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$

❖ All serializable schedules are not conflict-serializable but all conflict-serializable schedules are serializable

- **Precedence Graph/ Serialization graph/ Conflict graph:** a directed graph where the vertices are the transactions (names); we draw an arc from Ti to Tj if the two transactions conflict, and Ti accessed the data item on which the conflict arose earlier.



  - A schedule is conflict serializable if and only if its precedence graph is acyclic
  - If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph



- Consider the following schedule:
  - $w_1(A), r_2(A), w_1(B), w_3(C), r_2(C), r_4(B), w_2(D), w_4(E), r_5(D), w_5(E)$
- We start with an empty graph with five vertices labeled T_1, T_2, T_3, T_4, T_5.
- We go through each operation in the schedule:
  $w_1(A)$: A is subsequently read by T_2, so add edge $T_1 \rightarrow T_2$
  $r_2(A)$: no subsequent writes to A, so no new edges
  $w_1(B)$: B is subsequently read by T_4, so add edge $T_1 \rightarrow T_4$
  $w_3(C)$: C is subsequently read by T_2, so add edge $T_3 \rightarrow T_2$
  $r_2(C)$: no subsequent writes to C, so no new edges
  $r_4(B)$: no subsequent writes to B, so no new edges
  $w_2(D)$: C is subsequently read by T_2, so add edge $T_3 \rightarrow T_2$
  $w_4(E)$: E is subsequently written by T_5, so add edge $T_4 \rightarrow T_5$
  $r_5(D)$: no subsequent writes to D, so no new edges
  $w_5(E)$: no subsequent operations on E, so no new edges
- We end up with precedence graph
- This graph has no cycles, so the original schedule must be serializable. Moreover, since one way to topologically sort the graph is $T_3 - T_1 - T_4 - T_2 - T_5$, one serial schedule that is conflict-equivalent is
  - $w_3(C), w_1(A), w_1(B), r_4(B), w_4(E), r_2(A), r_2(C), w_2(D), r_5(D), w_5(E)$

View Serializability:

- **View Equivalent:** Let S and S' be two schedules with the same set of transactions. S and S' are view equivalent if the following three conditions are met, for each data item Q,
  - **Initial Read:** If in schedule S, transaction Ti reads the initial value of Q, then in schedule S' also transaction Ti must read the initial value of Q

    - **Write-Read Pair:** If in schedule S transaction Ti executes read(Q), and that value was produced by transaction Tj (if any), then in schedule S' also transaction Ti must read the value of Q that was produced by the same write(Q) operation of transaction Tj
    - **Final Write:** The transaction (if any) that performs the final write(Q) operation in schedule S must also perform the final write(Q) operation in schedule S'
- View serializable: a schedule S is view serializable if it is view equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable
- Below is a schedule which is view-serializable but not conflict serializable:

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- **Blind write:** performing a write operation without having performed a read operation
- Every view serializable schedule that is not conflict serializable has blind writes

Recoverability:

- Watch/look at lecture or ppt

Transaction Control Language (TCL):

- COMMIT: to save the changes

```
DELETE FROM Customers WHERE AGE = 25;
COMMIT;
```

- ROLLBACK: to roll back the changes

```
SQL> DELETE FROM Customers WHERE AGE = 25;
SQL> ROLLBACK;
```

- SAVEPOINT: creates points within the groups of transactions in which to ROLLBACK

```
SAVEPOINT SP1;
DELETE FROM Customers WHERE ID=1;
SAVEPOINT SP2;
DELETE FROM Customers WHERE ID=2;
SAVEPOINT SP3;
DELETE FROM Customers WHERE ID=3;
```

- The syntax for rolling back to a SAVEPOINT is:

```
ROLLBACK TO SAVEPOINT_NAME;
```

- The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

- SET TRANSACTION: places a name on a transaction
- Transactional control commands are only used with the DML Commands such as INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both: conflict serializable and recoverable (preferably, cascadeless)
- **Goal:** To develop concurrency control protocols that will assure serializability

- One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item
- The most common method used to implement locking requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item

Lock-Based Protocols:

- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks
- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes:
    - Exclusive (X) mode: data item can be both read as well as written; X-lock is requested using lock-X instruction
    - Shared (S) mode: data item can only be read; S-lock is requested using lock-S instruction
- A transaction can unlock a data item Q by the unlock(Q) Instruction
- Lock requests are made to the concurrency-control manager by the programmer
- Transaction can proceed only after request is granted
- **Lock-Compatibility Matrix:** a lock compatibility matrix is used which states whether a data item can be locked by two transactions at the same time

State of the lock \ Lock request type	Shared	Exclusive
Unlock	Yes	Yes
Shared	Yes	No
Exclusive	No	No

- **Requesting for / Granting of a Lock:** a transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- **Sharing a Lock:** any number of transactions can hold shared locks on an item unless a transaction holds an exclusive lock on the item no other transaction may hold any lock on the item
- **Waiting for a Lock**: if a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released
- **Holding a Lock:** a transaction must hold a lock on a data item as long as it accesses that item
- **Unlocking / Releasing a Lock:** transaction Ti may unlock a data item that it had locked at some earlier point; it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured

Two-Phase Locking Protocol:

- This protocol ensures conflict-serializable schedules
- **Phase 1: Growing Phase**
    - Transaction may obtain locks
    - Transaction may not release locks

  Phase 2: Shrinking Phase
    - Transaction may release locks
    - Transaction may not obtain locks
- **Lock point:** the point where a transaction acquired its final lock
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points

- **Deadlocks:** system is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set
- **Starvation/Livelock:** the same transaction is repeatedly rolled back due to deadlocks
- Cascading roll-back is possible under two-phase locking
- S**trict two-phase locking**: a transaction must hold all its exclusive locks till it commits/aborts; avoids cascading roll-back
- **Rigorous two-phase locking:** *all locks* are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit
- Concurrency goes down as we move to more and more strict locking protocol

Implementation of Locking:

- **A lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests

Deadlock handling:

- **Deadlock Prevention** protocols ensure that the system will never enter into a deadlock state
- Some prevention strategies:
    - Require that each transaction locks all its data items before it begins execution (pre-declaration)
    - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order
- **Transaction Timestamp:** Timestamp is a unique identifier created by the DBMS to identify the relative starting time of a transaction. Timestamping is a method of concurrency control in which each transaction is assigned a transaction timestamp
- Following schemes use transaction timestamps for the sake of deadlock prevention alone:
    - wait-die scheme (non-pre-emptive): older transaction may wait for younger one to release data item; younger transactions never wait for older ones, they are rolled back instead. A transaction may die several times before acquiring needed data item
    - wound-wait scheme (pre-emptive): older transaction wounds (forces rollback) of younger transaction instead of waiting for it; younger transactions may wait for older ones. May be fewer rollbacks than wait-die scheme
- Timeout-Based Schemes:
    - A transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted
    - Thus, deadlocks are not possible
    - Simple to implement; but starvation is possible. Also, difficult to determine good value of the timeout interval
- The system is in a deadlock state if and only if the wait-for graph has a cycle

Timestamp-Based Protocols:

- The protocol manages concurrent execution such that the time-stamps determine the serializability order
- In order to assure such behaviour, the protocol maintains for each data Q two timestamp values:
    - W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully

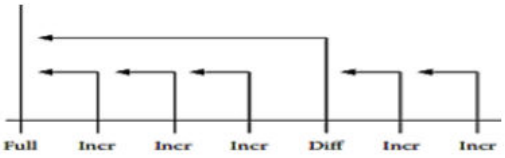
    - R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully
- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits but the schedule may not be cascade-free, and may not even be recoverable

Backup

- A **Backup** of a database is a representative copy of data containing all necessary contents of a database such as data files and control files
- **Physical Backup:** A copy of physical database files such as data, control files, log files, and archived redo logs
- **Logical Backup:** A copy of logical data that is extracted from a database consisting of tables, procedures, views, functions, etc
- **Necessity for a backup**: disaster recovery, client-side changes, auditing, downtime
- **Types of backup data**:
    - Business Data includes personal information of clients, employees, contractors etc. along with details about places, things, events and rules related to the business
    - System Data includes specific environment/configuration of the system used for specialised development purposes, log files, software dependency data, disk images
    - Media files like photographs, videos, sounds, graphics etc. need backing up. Media files are typically much larger in size

<u>Backup Strategies:</u>

- **Full Backup:** stores all the objects of the database; tables, procedures, functions, views, indexes etc.
    - Advantages: independent backups; easy to setup, configure and maintain
    - Disadvantages: time, system downtime, large storage needed (all relative to other backups)
- **Incremental Backup:** targets only those files or items that have changed since the last backup
    - Anything with a greater timestamp than previous backup is backed up
    - This ensures a minimum backup window during peak activity times, with a longer backup window during non-peak activity times
    - Advantages: less storage per backup, downtime is minimised, cost reduction compared to full backup
    - Disadvantages: more effort to recover
- **Differential Backup:** backup backs up all the changes that have occurred since the most recent full backup regardless of what backups have occurred in between



    - Recovery on any given day only needs the data from the full backup and the most recent differential backup
    - Advantages: recoveries require fewer backup sets, better recovery options
    - Disadvantages: storage may exceed incremental backups, can even reach the size of a full backup if don't after a long time

- **Hot backup:** refers to keeping a database up and running while the backup is performed concurrently
    - Preferable whenever high availability is a requirement
    - Dynamic data and systems which run 24x7 need hot backups
    - Advantages: high availability of database, easier point-in-time recovery, best for dynamic and modularized data
    - Disadvantages: may not be feasible when the data set is huge and monolithic, low fault tolerance, high cost of maintenance and setup
    - Mainly used for **Transaction Log Backup**

RAID: Redundant Array of Independent Disks (IMPORTANT):

- Disk organization techniques that manage a large number of disks, providing a view of a single disk of
    - high capacity and high speed by using multiple disks in parallel,
    - high reliability by storing data redundantly, so that data can be recovered even if a disk fails
- Techniques sued to achieve RAID levels:
    - Redundancy
    - Mean time to data loss, mean time to repair
    - Mirroring/shadowing: duplicate every disk, logical disk consists of two physical disks
    - Bit-level Striping: split the bits of each byte across multiple disks
    - Byte-level Striping: each file is split up into parts one byte in size. Using n = 4 disk array as an example
    - Block-level Striping (most common): with n disks, block i of a file goes to disk (i mod n) + 1
    - Bit-Interleaved Parity: a single parity bit is enough for error correction, not just detection, since we know which disk has failed
    - Block-Interleaved Parity: uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from n other disks

- **RAID 0 (Striping):**
    - RAID level-0 only uses data striping, no redundant information is maintained
    - Effective space utilization for a RAID Level-0 system is always **100 percent**
    - best write performance
    - least costly
    - Reliability is very poor

- **RAID 1 (Mirroring):**
    - RAID 1 employs mirroring, maintaining two identical copies of the data on two different disks
    - most expensive solution
    - excellent fault tolerance
    - parallel reads
    - transfer rate for a single request is comparable to the transfer rate of a single disk
    - The effective space utilization is **50 percent**

- **RAID 2 (Parity):**
    - RAID 2 uses designated drive for parity
    - In RAID 2, the striping unit is a single bit
    - Hamming code

- Effective disks utilised = (n + 1)/ n

- **RAID 3 (Byte Striping + Parity):**
  - single check disk with parity information
  - cannot service multiple requests simultaneously

- **RAID 4 (Block Striping + Parity):**
  - striping unit of a disk block
  - Provides recovery of corrupted or lost data using XOR recovery mechanism
  - Facilitates recovery of at most 1 disk failure
  - Write performance is low due to the need to write all parity data to a single disk
  - 1 block is parity block

- **RAID 5 (Distributed Parity):**
  - improves upon RAID 4 by distributing the parity blocks uniformly over all disks instead of storing them on a single check disk
  - This level too allows recovery of only 1 disk failure like level 4

- **RAID 6 (Dual Parity):**
  - RAID 6 extends RAID 5 by adding another parity block, thus it uses block-level striping with two parity blocks distributed across all member disks
  - Write performance of RAID 6 is poorer than RAID 5
  - disk failure during recovery of a failed disk can be handled

Comparison of RAID: Theoretical

PPD

Level	Description	Min.[b] # of drives	Space Efficiency	Fault Tolerance (*Drives*)	Performance	
					Read	Write
					(*as factor of single disk*)	
RAID 0	Block-level striping without parity or mirroring	2	1	None	n	n
RAID 1	Mirroring without parity or striping	2	$\frac{1}{n}$	$n-1$	$n^{[a]}$	$1^{[c]}$
RAID 2	Bit-level striping with Hamming code for error correction	3	$1-\frac{1}{n}\lg(n+1)$	One[d]	Depends	Depends
RAID 3	Byte-level striping with dedicated parity	3	$1-\frac{1}{n}$	One	$n-1$	$n-1^{[e]}$
RAID 4	Block-level striping with dedicated parity	3	$1-\frac{1}{n}$	One	$n-1$	$n-1^{[e]}$
RAID 5	Block-level striping with distributed parity	3	$1-\frac{1}{n}$	One	$n^{[e]}$	single sector: $\frac{1}{4}$ full stripe: $n-1^{[e]}$
RAID 6	Block-level striping with double distributed parity	4	$1-\frac{2}{n}$	Two	$n^{[e]}$	single sector: $\frac{1}{6}$ full stripe: $n-2^{[e]}$

[a] Theoretical maximum, as low as single-disk performance in practice
[b] Assumes a non-degenerate minimum number of drives
[c] If disks with different speeds are used in a RAID 1 array, overall write performance is equal to the speed of the slowest disk
[d] RAID 2 can recover from one drive failure or repair corrupt data or parity when a corrupted bit's corresponding data and parity are good
[e] Assumes hardware capable of performing associated calculations fast enough
Source: *Standard RAID levels* (Accessed 23-Aug-2021)

Comparison of RAID: Practical

PPD

Features	RAID 0	RAID 1	RAID 5	RAID 6	RAID 10
Minimum # of drives	2	2	3	4	4
Fault tolerance	None	Single-drive failure	Single-drive failure	Two-drive failure	Up to 1 disk failure in each sub-array
Read performance	High	Medium	Low	Low	High
Write Performance	High	Medium	Low	Low	Medium
Capacity utilization	100%	50%	67% – 94%	50% – 88%	50%
Typical applications	*High end workstations, data logging, real-time rendering, very transitory data*	*Operating systems, transaction databases*	*Data warehouse, web servers, archiving*	*Data archive, backup to disk, high availability solutions, servers with large capacity requirements*	*Fast databases, file servers, application servers*

Source: *RAID Level Comparison: RAID 0, RAID 1, RAID 5, RAID 6 and RAID 10* (Accessed 23-Aug-2021)

- RAID does not equate to 100% uptime: Nothing can. RAID is another tool on in the toolbox meant to help minimize downtime and availability issues. There is still a risk of a RAID card failure, though that is significantly lower than an HDD failure
- RAID does not replace backups: Nothing can replace a well-planned and frequently tested backup implementation!

- RAID does not protect against data corruption, human error, or security issues: While it can protect you against a drive failure, there are innumerable reasons for keeping backups. So, RAID is not a replacement for backups
- RAID does not necessarily allow to dynamically increase the size of the array: If you need more disk space, you cannot simply add another drive to the array. You are likely going to have to start from scratch, rebuilding/reformatting the array. Luckily, Steadfast engineers are here to help you architect and execute whatever systems you need to keep your business running.
- RAID isn't always the best option for virtualization and high-availability failover: You will want to look at SAN solutions

Recovery

- **Recovery** is the process of restoring the database to its latest known consistent state after a system failure occurs
- A **Database Log** records all transactions in a sequence. Recovery using logs is quite popular in databases. A typical log file contains information about transactions to execute, transaction states, and modified values
- Failure Classifications: transaction failure (logical and system errors), system crash, disk failure
- Recovery algorithms have two parts
    - Actions taken during normal transaction processing to ensure enough information exists to recover from failures
    - Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

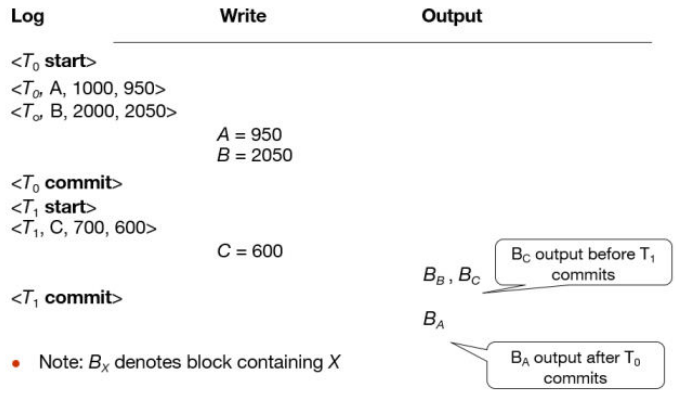
Storage Structure:

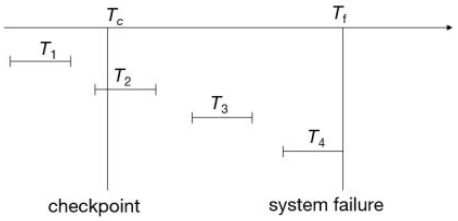
- **Volatile Storage:** does not survive system crashes, examples: main memory, cache memory
- **Nonvolatile Storage:** survives system crashes, examples: disk, tape, flash memory, non-volatile (battery backed up) RAM; but may still fail, losing data
- **Stable Storage:** a mythical form of storage that survives all failures, approximated by maintaining multiple copies on distinct non-volatile media
- System Buffer blocks are those blocks residing temporarily in main memory.
- Physical blocks are the blocks residing on the disk

Log-Based Recovery/ Shadow Paging:

- The log is a sequence of log records, which maintains information about update activities on the database
- A log is kept on stable storage
- On a log, a transaction registers itself with <Ti start> when it starts; before a transaction executes, a log record of <Ti, X, V1, V2> is written where V1 is the value of X before the write (old value), and V2 is the value to be written to X (new value); when Ti finishes its last statement, the log record < Ti commit > is written
- **The immediate-modification scheme** allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits

- Example:



  - **Undo** of a log record < Ti, X, V1, V2 > writes the old value V1 to X; when undo of a transaction is complete, a log record < Ti abort> is written out; the undo is used for transaction rollback during normal operation
  - **Redo** of a log record < Ti, X, V1, V2 > writes the new value V2 to X; no logging done
  - The undo and redo operations are used during recovery from failure
  - Write a log record < checkpoint L > onto stable storage where L is a list of all transactions active at the time of checkpoint
  - Any transactions that committed before the last checkpoint should be ignored
  - Any transactions that committed since the last checkpoint need to be redone
  - Any transaction that was running at the time of failure needs to be undone and restarted
  - 

  - In this example, T1 can be ignored, T2 and T3 need to be redone, T4 needs to be undone and restarted
- **The deferred-modification scheme** performs updates to buffer/disk only at the time of transaction commit
- A transaction is said to have committed when its commit log record is output to stable storage; all previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Query Processing