Database Management Systems

WEEK 1

→ Drawbacks of using file systems to store data:
- ↳ Data redundancy and inconsistency
  - → multiple file formats ; duplication of information
- ↳ Difficulty in accessing data
- ↳ Data isolation
- ↳ Data integrity
- ↳ Atomicity of updates
  - → failures may leave databases in an inconsistent state with partial data updates carried out.
- ↳ Concurrent access by multiple users
- ↳ Security problems

	Bookkeeping	Spreadsheet files
Durability	Physical damage	computer applications
Scalability	difficult to maintain years of data	easier to search, insert and modify
Security	Susceptible to tampering	can be password-protected
Retrieval	Time consuming to search	searching and manipulation requires less manpower
Consistency	Prone to human errors	less prone to mistakes

→ Client/server RDBMS
- ↳ SYBASE
- ↳ ORACLE DATABASE
- ↳ SQL SERVER
- ↳ INFORMIX
- ↳ MY SQL

	File handling using Python	DBMS
Scalability (amount of data)	difficult to insert, update and querying of records.	in-built feature to provide high scalability.
Scalability (changes in structure)	extremely difficult to add or remove attributes.	adding/removing attributes seamless using simple SQL queries
Time of execution	in seconds	in milliseconds
Persistence	processing using temp. data structures that have to be manually updated.	automatic, system induced mechanisms
Robustness	has to be done manually	backup, recovery, and restore need minimum manual intervention
Security	difficult to implement in Python	user-specific access at database level.
Programmer's productivity	most operations require extensive code to ensure persistence, robustness, and security	standard and simple in-built queries reduce the effort
Arithmetic operations	easy to do arithmetic computations	limited set of arithmetic operations
Costs	low costs for hardware, software and labour	high costs for hardware, software, labour

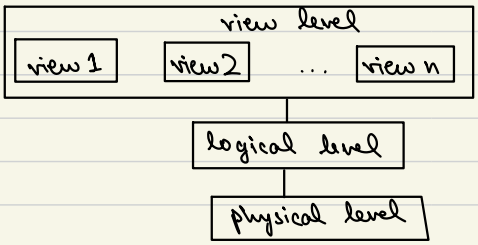
→ Levels of Abstraction

① Physical level → describes how a record is stored

② Logical level → " data stored in database; relationships among data and fields.

```
type instructor = record
    ID : string;
    name : string;
    dept_name : string;
    salary : integer;
end;
```

③ View level → application programs hide details of data types



→ Schema and Instances

→ schema: the way data is organised; instance: actual data

→ logical schema: analogous to type information of a variable

↳ customer schema:

Name	Customer ID	Account #	Aadhaar ID	Mobile #

→ physical schema: physical structure

→ instance: actual content; analogous to value of variable

↳ customer instance

Name	Customer ID	Account #	Aadhaar ID	Mobile #
Pavan Laha	6728	917322	182719289372	9830100291
Lata Kala	8912	827183	918291204829	7189203928
Nand Prabhu	6617	372912	127837291021	8892021892

→ physical data independence: ability to modify the physical schema without changing the logical schema

→ Data Models
- → Relational Model → tabular form
- → Entity-Relationship data model (for database design)
- → Object-based data models (object-oriented and object-relational)

→ DDL (Data Definition Language)
- → the way to deal with the schema
  - → defines schematic of database
- → DDL compiler generates a set of table stored in data dictionary

→ DML (Data Manipulation Language) a.k.a. Query Language
- → for accessing and manipulating data
- → classes:
  - ① Pure → Relational Algebra
    - → Tuple relational calculus
    - → Domain " "
  - ② Commercial → SQL

→ SQL → usually embedded in some higher-level language to be able to do complex functions.

→ Database Design
- → Logical Design → deciding on schema
  - → business decision: what attributes?
  - → comp. science decision: what relational schema?
- → Physical Design → how you physically lay out data | files | etc.

Design Approaches
- ① Entity Relationship Model (Ch. 7)
  - ↳ business requirements
- ② Normalization Theory (Ch. 8)

Object Relational Data Model
- → relational model stores atomic values
- → object relational data model stores objects

XML: Extensible Markup Language
- → ability to specify new tags, and to create nested tag structures
- → basis for all new generation data interchange formats

→ Database Engine

→ Storage Manager

↳ stores data ; physical design , data dictionary

↳ responsible for · - interaction with OS file manager

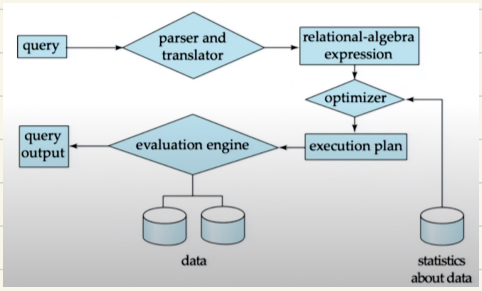
- efficient storing , retrieving and updating data

→ Query Processing

① Parsing and translation

② Optimization

③ Evaluation



→ Transaction Manager

→ transaction : collection of operations that performs in a single logical function in DB application

→ transaction - management component : ensures DB remains in a consistent state despite failures

→ concurrency - control manager : controls the interaction among concurrent transactions to ensure consistency of DB

WEEK 2

→ Attributes

→ normally required to be atomic
→ null value is a member of every domain
→ Domain of attributes: defines the type. e.g.. Name : string
Adhaar ID : 12-digit number

→ Schema and Instance

→ $A_1, A_2, \ldots, A_n$ are attributes
$R = (A_1, A_2, \ldots, A_n)$ is a relational schema
$D_1, D_2, \ldots, D_n$ are domains of attributes
→ a relation is set of n-tuples $(a_1, a_2, \ldots, a_n)$, where $a_i \in D_i$

→ Order of tuples / row is irrelevant
No two tuples / row may be identical

→ Key

→ Let $K \subseteq R$, where R is the set of attributes.
→ K is a superkey of R if values of K are sufficient to identify a unique tuple of each possible relation r(R).
→ Superkey K is candidate key if K is minimal
↳ no subset of K is a superkey
→ Primary key → one of selected candidate key.
→ Surrogate key → synthetic key ; not derived from application data.
→ unique identifier for either an entity in modeled world or an object in database.

→ Secondary key → non-primary candidate keys
→ Simple key → consists of a single attribute
→ Composite key → " " multiple "
→ Foreign key → when an attribute is a key in a different table

→ Relational Query Languages

① Procedural programming
↳ programmer tells the computer what to do
↳ HOW to get the output

② Declarative programming
↳ more descriptive
↳ WHAT relationship holds b/w various entities
↳ useful for relational query

→ Select Operation

→ to select a subset of rows from a relational substance that satisfies a condition

→ e.g.

Relation r =

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D>5}(r)$ =

A	B	C	D
α	α	1	7
β	β	23	10

↳ A=B AND D>5

→ Project Operation

→ to select a few columns

e.g.

r =

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

$\Pi_{A,C}(r)$ =

A	C
α	1
~~α~~	~~1~~
β	1
β	2

=

A	C
α	1
β	1
β	2

→ Union of two relations

→

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r ∪ s =

A	B
α	1
α	2
β	1
β	3

→ Difference of two relations

→

r − s =

A	B
α	1
β	1

→ eliminate those things that are in s that are already in r.

→ Intersection of two relations

→

r ∩ s =

A	B
α	2

→ Cartesian-product

→

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

r × s =

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

→ Renaming a table

→ refer to a relation (say E) by more than one name

↳ $\rho_x(E)$ returns the expression E under the name X.

r =

A	B
α	1
β	2

r

$r \times \rho_s(r)$ =

r.A	r.B	s.A	s.B
α	1	α	1
α	1	β	2
β	2	α	1
β	2	β	2

↑ cartesian with itself

→ Composition of Operations

→ e.g. $\sigma_{A=C}(r \times s)$

$(r \times s)$ =

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

$\sigma_{A=C}(r \times s)$ =

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b

→ Natural Join

→ natural join of relations R and S is a relation on schema $R \cup S$:

→ t_r, t_s : tuples from r and s

→ if t_r and t_s have same values on each of the attributes $R \cap S$ (common attributes), add a tuple t to result, where:

t has same value as t_r on r

t " " " " t_s " s

→ example:

Relations =

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

$r \bowtie s$ =

A	B	C	D	E	
α	1	α	a	α	→ row 1 r, row 1 s
α	1	α	a	γ	→ row 1 r, row 3 s
α	1	γ	a	α	→ row 4 r, row 1 s
α	1	γ	a	γ	→ row 4 r, row 3 s
δ	2	β	b	δ	→ row 5 r, row 4 s

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

→ Aggregate Operators

→ SUM, AVG, MAX, MIN

→ SQL

History → developed by IBM

→ pronounced SEQUEL

Data Definition Language (DDL)

→ Schema for each relation

→ Domain of values of each attribute

→ Integrity constraints

→ relation defined using create table command:

create table $r(A_1 D_1, A_2 D_2, \ldots, A_n D_n)$,
(integrity-constraint$_1$),
...
(integrity-constraint$_k$);

A_i: attribute name in schema of relation r
D_i: data type of values in the domain of the attribute
r: name of relation

example:

- not null
- primary key ($A_1, \ldots, A_n$)
- foreign key ($A_m, \ldots, A_n$) references r

```
create table instructor (
    ID char(5),
    name varchar(20)
    dept_name varchar(20)
    salary numeric(8, 2));
```

```
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

→ Update tables
- → Insert (DML): add a value
- → Delete (DML): remove all rows from table
- → Drop Table (DDL): discard the entire table
- → Alter (DDL): add attribute or drop attribute

Data Manipulation Language (DML)

→ Typical SQL Query:

```
select attributes
from relations
where condition (Boolean)
```

→ Select → corresponds to projection operation of relational algebra
- → SQL names are case-insensitive
- → allows duplicates in relations and query results
  - ↳ use distinct to remove duplicates
- → select * → selects all attributes
- → can contain arithmetic operations (+, −, *, /)

→ Where → condition, True or False
- → corresponds to selection operation of relational algebra

→ From → scope of which relations you're dealing with
- → for multiple relations, by default, uses Cartesian product

→ String Operation
- → percent (%) → character matches any substring
- → underscore (_) → character matches any character

→ Ordering display of tuples; e.g. →

```
select distinct name
from instructor
order by name desc
```

→ Select top clause ; e.g. → select top 10 distinct name from instructor

→ Set operations

→ union, intersect, except

→ example:

- Find the salaries of all instructors that are less than the largest salary
  select distinct *T.salary*
  from *instructor* **as** *T*, *instructor* **as** *S*
  where *T.salary* < *S.salary*
  } will give highest salary
- Find all the salaries of all instructors
  select distinct *salary*
  from *instructor*
- Find the largest salary of all instructors
  (**select** "second query")
  except
  (**select** "first query")
  } will give all salaries except the highest salary

→ to retain all duplicates: union all, except all, intersect all

→ Null values

→ usually for when you dont know all the values of all the fields.

→ adding an attribute to a table.

→ the condition where is null can be used to check for null values. where is not null

→ any comparison with null returns unknown

→ three-valued logic

OR: (unknown or true) = true
(unknown or false) = unknown
(unknown or unknown) = unknown

AND: (unknown and true) = unknown
(unknown and false) = false
(unknown and unknown) = unknown

NOT: (not unknown) = unknown

→ (P is unknown) = true if P evaluates to unknown

→ if the result of condition is unknown, it is treated as false in the "where" statement.

Binary Numbers and Character Encoding

→ To convert a number into equivalent binary representation: divide the number by 2 until the quotient becomes 0. The remainder after every division (in the order) is the binary representation.

e.g. $(1729)_{10} \Rightarrow (11011000001)_2$

2 | 1729	
2 | 864	− 1
2 | 432	− 0
⋮	
2 | 13	− 1
2 | 6	− 1
2 | 3	− 0
2 | 1	− 1
0	− 1

$(2.718)_{10} \approx (10.10110)_2$

2 | 2	
2 | 1	− 0
0	− 1

$(2)_{10} = (10)_2$

0.718 × 2 = 1.436	1
0.436 × 2 = 0.872	0
0.872 × 2 = 1.744	1
0.744 × 2 = 1.488	1
0.488 × 2 = 0.976	0

$(0.718)_{10} = (10110)_2$ → not as precise because we didn't get to 0.

→ To convert from binary to number : $\sum_{i=0}^{n} x_i(2)^i$, where n = total # of digits , and i counts backward starting from the rightmost , and x_i = ith digit.

e.g. 1 1 0 1 1 0 0 0 0 0 1 → n
10 9 8 7 6 5 4 3 2 1 0 → i

$1*(2)^0 + 1*(2)^9 + 0*(2)^8 + \dots + 0*(2)^2 + 0*(2)^1 + 1*(2)^0 = 1729$

WEEK 3

→ Nested Subqueries

→ 'select - from - where' nested within another query.

→ As a part of WHERE clause :

→ typical use → set membership / comparison / cardinality

example :

Set Membership

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

```
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);
```

} nested subquery

→ SOME clause

→ F <comp> some r ⇔ ∃t ∈ r such that (F <comp> t)

↙ some t for which comparison is true

where <comp> can be : <, ≤, >, ≥, =, ≠

→ existential quantification

example :

Set Comparison – "some" Clause

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

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using **some** clause

```
select name
from instructor
where salary > some (select salary
                    from instructor
                    where dept_name = 'Biology');
```

→ ALL clause

→ F <comp> all r ⇔ ∀t ∈ r such that (F <comp> t)

↳ comparison is true for all.

→ universal quantification

→ EXISTS clause

→ returns true if argument subquery is nonempty

Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

```
select course_id
from section as S
where semester = 'Fall' and year = 2009 and
      exists (select *
              from section as T
              where semester = 'Spring' and year = 2010
              and S.course_id = T.course_id);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

→ UNIQUE clause

→ tests whether a subquery has any duplicates

example:

Find all courses that were offered at most once in 2009

```
select  T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id = R.course_id
              and R.year = 2009);
```

→ As a part of FROM clause

→ 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;
```

→ WITH clause

→ defining a temporary relation whose definition is only available to the query in which the with clause is.

example:

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;
```

creates a table with one attribute

subquery gives a value

basic query

example:

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at 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;
```

→ As a part of SELECT clause

→ needs scalar subquery

example:

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

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

→ Modifications of the Database

→ **Deletion** of a tuple

examples :

→ Delete all tuples : `delete from instructors`

→ Delete with condition : `delete from instructors where dept_name = 'Finance'`

→ **Insertion** of a tuple

examples :

→ Add a new tuple : `insert into course values ('CS-437', ..., 4)`

→ **Updation** of a tuple

Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%

- Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

CASE statement for conditional updates

↳

- Same query as before but with **case** statement

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

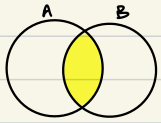
→ CROSS JOIN

→ returns cartesian product of rows

```
select *
from employee, department
```

→ INNER JOIN

→ where only the intersecting tuples are taken

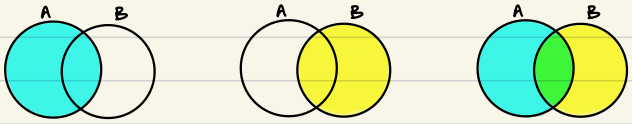


→ ignores uncommon tuples

→ natural join removes duplicity of common column

→ OUTER JOIN

→ LEFT or RIGHT or FULL



natural left/right/full outer join

→ VIEW

→ defined using `create view` statement:

`create view v as <query expression>`

creates a virtual relation with the name 'v'.

→ examples:

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
    from faculty
    where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

→ view recursive: a view relation that depends on itself to be defined.

→ insertion in a view is actually done by insertion in the relation.

→ Transactions

→ unit of work; atomic (fully executed or entirely rolled back)

→ ended by `commit work` or `rollback work`

→ Integrity constraints in a single relation

→ `not null`

↳ declare an attribute to not allow null values

→ `unique` $(A_1, A_2, \ldots, A_m)$

↳ the attributes form a candidate key.

→ `check`

↳ to impose business logic

↳ e.g. `check (semester in ('Fall', 'Winter', 'Spring'))`

→ Referential Integrity

→ 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:

`foreign key (dept_name) references department`

{ `on delete cascade`
{ `on update cascade`

↳ maintains integrity; if key deleted in foreign relation, then all tuples with that key will be deleted in this relation.

→ SQL Datatypes and Schemas
  → date, time, timestamp, interval

→ Index creation
  → create index index_name on relation(attribute)

→ User-Defined Types (UDT)
  → create type Dollars as numeric(12,2) final

→ Domains
  → create domain person_name char(20) not null
  → create domain degree_level varchar(10)
    constraint degree_level_test
    check (value in ('Bachelors', 'Masters', 'Doctorate'));

→ Authorisation
  → grant <privilege list>
    on <relation/view name> to <user list>

  → Privileges
    → select : read access
    → insert : ability to insert tuples
    → update : ability to use SQL update statement
    → delete : delete tuples
    → all privileges : for all allowable privileges

  → revoke → revokes the privileges

  → Roles
    ↳ creates a role ; then you grant privileges to roles and roles can be granted to users.
    ↳ create role <role_name>
    ↳ roles can be granted to other roles, creating a hierarchy of authorisation.

→ Function and Procedures

→ example:

SQL Functions

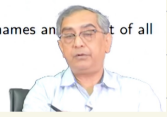
- Define a function that, given the name of a department, returns the count of the number of instructors in that department:

```
create function dept_count (dept_name varchar(20))
    returns integer
    begin
    declare d_count integer;
        select count (*) into d_count
        from instructor
        where instructor.dept_name = dept_name
    return d_cont;
    end
```

Local var

- 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
```



→ function is a parameterised view

→ Table functions

↳ returns a table

↳ example:

- **Functions that return a relation as a result** added in SQL:2003
- 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) )
    returns table
        (select ID, name, dept_name, salary
        from instructor
        where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *
from table (instructor_of ('Music'))
```

→ Procedures

↳ do not return anything. They have 'in' parameters and 'out' parameters.

↳ example:

- 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
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

to execute a procedure

- Procedures and functions can be invoked also from dynamic SQL
- SQL:1999 allows **overloading** - more than one function/procedure of the same name as long as the number of arguments and / or the types of the arguments differ

→ Language Constructs

→ begin ... end → compound statements

→ while boolean expression do
    sequence of statements;
end while;

→ repeat
    sequence of statements;
until boolean expression
end repeat;

→ Find the budget of all departments:

```
declare n integer default 0;
for r as
        select budget from department
    do
        set n = n + r.budget
    end for;
```

→ if - then - else

→ case variable

```
when value1 then
        sequence of statements
when value2 then
        sequence of statements
...
else
        sequence of statements
end case;
```

→ searched case example:

Language Constructs (6): Searched case

- Searched **case** statement

```
case
        when sql-expression = value1 then
                sequence of statements;
        when sql-expression = value2 then
                sequence of statements;
        ...
        else
                sequence of statements;
end case;
```

→ Triggers

→ set of actions performed in response to an insert, delete, or update operation.

→ create trigger statement

→ Statement level triggers
↳ applies to all rows of the statement result

→ Row Level triggers
↳ applies to a single row

→ example:

Trigger to Maintain credits_earned value

```
create trigger credits_earned after update of grade on takes
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <>'F' and nrow.grade is not null
        and (orow.grade = 'F' or orow.grade is null)
begin atomic
        update student
        set tot_cred= tot_cred +
            (select credits
            from course
            where course.course_id=nrow.course_id)
        where student.id = nrow.id;
end;
```

WEEK 4

→ Formal relational query language systems :

① Relational algebra → procedural and algebra based
② Tuple relational calculus → non-procedural and predicate calculus
③ Domain relational calculus → " " " "

→ Relational Algebra

→ basic operators : select : σ set difference : $-$ and : $\wedge$
project : Π cartesian product : $\times$ or : $\vee$
union : $\cup$ rename : ρ not : $\neg$

→ Division

→ example :

R

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

S

Subject
Databases
Prolog

R | S

Lecturer
Green

'Green' is associated with all instances in S relation.

↳ needs one common attribute.
↳ looks for (R-S) attributes that have the same values on common attribute.
↳ in the example, 'Brown' Lecturer didn't have an entry for 'Prolog' Modules so disqualified. 'Green' Lecturer is the only value that has the same value for both 'Prolog' and 'Databases'.

→ example :

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

→ Predicate Logic

- → extension of propositional logic / boolean algebra
- → Predicate is a function.
  - ↳ example: $P(x) = x > 3$
  - ↳ once you assign a value to 'x', it becomes a proposition and has a truth/false value.
  - ↳ can have multiple variables: $P(x_1, x_2, x_3, \ldots, x_n)$

- → Quantifiers → used to talk about properties that hold over the entire domain.
  - ↳ Universal quantifiers → has to hold true for all elements in the domain.
    - ↳ example: $\forall x\, P(x)$ read as "for all x P(x)"
      - ↳ $P(x) = x + 2 > x$; then $\forall x\, P(x)$ is True
  - ↳ Existential quantifiers → has to hold for at least one element
    - ↳ example: $\exists x\, P(x)$
      - ↳ $P(x) = x > 5$; then $\exists x\, P(x)$ is True

→ Tuple Relational Calculus

- → $\{t \mid P(t)\}$
- → example: find first names of all students with age > 21.
  - ↳ $\{t.Fname \mid t \in Student(t) \wedge t.age > 21\}$

→ Domain Relational Calculus

- → no diff. from tuple relational other than notation.

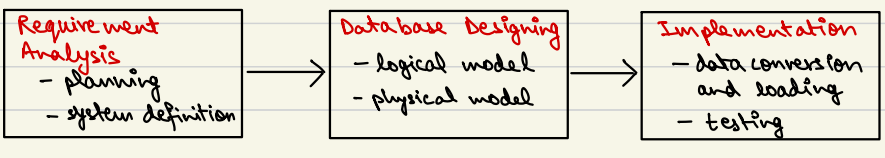
→ Equivalence of RA, TRC, DRC

- → example: select operation on a relation R = (A, B)
  - RA: $\sigma_{B=17}(r)$
  - TRC: $\{t \mid t \in r \wedge t.B = 17\}$
  - DRC: $\{\langle A, B\rangle \mid \langle A, B\rangle \in r \wedge B = 17\}$

→ Design of an E-R model

- → satisfies functional specification
- → conforms to limitations
- → meets requirements on performance and resource usage
- → satisfies design criteria
- → " restrictions on the design process

- → Two components: ① Abstraction
  ② Model



→ Abstraction of a design
  → abstract things down so we have the information but it needs a lot less memory | time | effort.
  → example: when dealing with courses | grades, salary of the instructor is irrelevant.

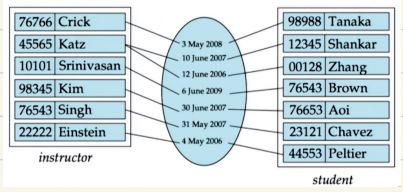
→ Models of a design
  → models of the real world that mimic the real world.

→ Entity - Relationship Model
  → Framework for logical model
  → Conceptual Requirements

  → Attributes
    ↳ properties associated with an entity | entity set
    ↳ simple and composite < first name / last name
    ↳ single - valued or multivalued
      ↳ example: multiple cars of a single person
    ↳ derived attributes
      ↳ example: age, given DOB
    ↳ domain → set of permitted values

  → Entity sets
    ↳ represented by a list of attributes
    ↳ relations | tables

  → Relationship sets
    ↳ association among several entities.
      → example: 44553 (Peltier) — advisor relationship set — 22222 (Einstein)
        student entity — instructor entity
        ↳ (44553, 22222) ∈ advisor
    ↳ relationship could itself have an attribute
      example:



    ↳ Mapping cardinalities
      → One - to - One
      → One - to - Many
      → Many - to - One
      → Many - to - Many

→ **Weak Entity Set**

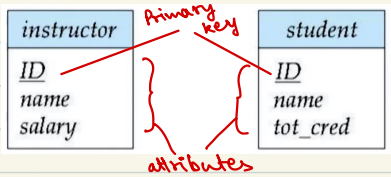
→ strong entity sets have a primary key.

→ weak " " have a partial key, set of (discriminator) attributes that uniquely identify a group of entities.

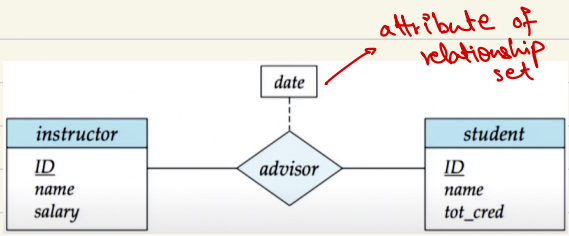
→ They feature in relationship with a strong entity set

→ **ER diagram**

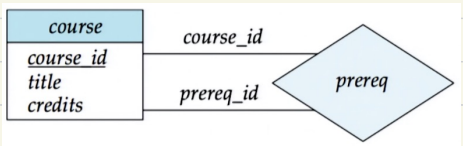
→ Entity Sets
↳ Box



→ Relationship Sets
↳ Diamond



→ Roles
↳ entity set may have relationship with itself.

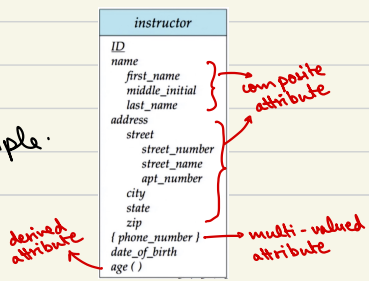


→ Cardinality constraints
↳ one-to-many and so on.
↳ directed line (⟶) means "one".
undirected " (——) means "many".

→ Total | Partial participation
↳ Total : (═══) every tuple participates in the relationship.
↳ Partial : (——) not every tuple participates.

→ Complex cardinality
↳ min | max cardinality shown in the form.
l...h , where l = min and h = max

→ Complex attributes
↳ shown with a tab.
example.

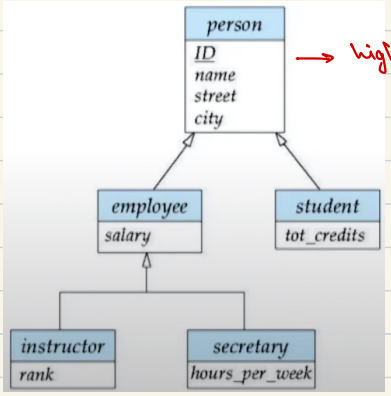


→ Weak entity sets
  ↳ Double-lined boundary
  ↳ diamond (relationship) will also be double-lined.
  ↳ discriminator underlined with dashed

→ ER model extended features

→ Specialisation → top-to-bottom design
  → ISA
    ↳ if r_1 ISA r_2, then r_1 has all the attributes of r_2. Depicted by (⟶▷)

Overlapping : employee and person
Disjoint : instructor and secretary



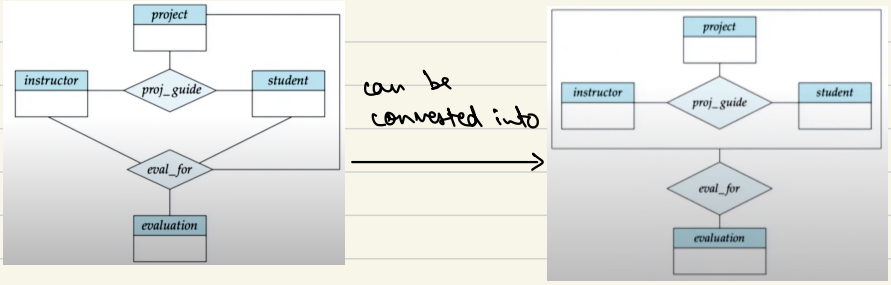
→ highest entity set

- If it is not necessary that every entity in a higher entity set exists in one or more lower entity sets, then the specialisation is partial otherwise total.

→ Generalisation → bottom-to-top design

↳ tends to be total but not always the case.

→ Aggregation



→ proj_guide can be treated as an entity, to remove redundancy.
→ eval_for is: eval_for(s_ID, project_id, i_ID, evaluation_id)

X X X

WEEK 5

→ Features of a good relational design

- → reflects real-world structure
- → can represent all expected data over time
- → avoids redundant storage of data
- → provides efficient access to data
- → supports maintenance of data integrity
- → clean, consistent, and easy to understand

→ Redundancy → multiple copies of same data ; leads to anomaly

→ Anomaly → inconsistencies due to changes in database

- → Insertion anomaly
  - ↳ happens when insertion of a data record is not possible without adding some additional unrelated data.
- → Deletion anomaly
  - ↳ when deletion of a record results in losing some unrelated information
- → Update anomaly
  - ↳ when a data is changed, many records have to be changed leading to possibility of some changes being made incorrectly.

→ To minimise redundancy, minimise dependency

to " dependency, you need a good decomposition of the database.

→ Lossless Join Decomposition

- → a decomposition of a relation R into relations R_1, R_2 such that if we perform natural join of R_1 and R_2, it will return R.

→ Atomic Domains and First Normal Form

- → a domain is atomic if its elements are indivisible units.
  - ↳ example of non-atomic :- set of names
    - id numbers like CS101
- → a relational schema R is in First Normal Form (1NF) if
  - → domains of all attributes are atomic.
  - → every attribute contains only a single-value.

→ Functional Dependencies

→ value of certain attributes determining uniquely the value for another set of attributes.

→ $\alpha \subseteq R$ and $\beta \subseteq R$

$\alpha \rightarrow \beta$ holds on R iff $\forall$ legal relation $r(R)$, for any two tuples t_1 and t_2 ; if $t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$

→ K is a **superkey** for relational schema R iff $K \rightarrow R$

K is a **candidate key** for R iff:

$K \rightarrow R$; and

for no $\alpha \subset K$, $\alpha \rightarrow R$

→ Reflexivity : if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$

Augmentation : if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$

Transitivity : if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$

} Axioms

→ Axioms can be applied to generate new FDs

↳ a new FD obtained by applying the axioms is **logically implied**

↳ Axioms are:

Sound : generate only FDs that actually hold

Complete : eventually generate all FDs that hold

→ Closure set of FDs $F = F^+$

↳ set of all FDs logically implied by F

→ Union : if $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta\gamma$

Decomposition : if $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$

Pseudotransitivity : if $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$

} derived rules

→ **Closure of attribute sets**

→ R = (A, B, C, G, H, I) ; F = {A → B, A → C, CG → H, CG → I, B → H}

→ $(AG)^+$ steps:

① AG

② ABCG because (A → C and A → B)

③ ABCGH because (CG → H and CG ⊆ ABCG)

④ ABCGHI because (CG → I and CG ⊆ ABCGH)

→ $(AG)^+$ is a super key because $(AG)^+$ has all attributes of R.

→ $(AG)^+$ is minimal because $(A)^+ = (ABCH)$ and $(G)^+ = (G)$

hence $(AG)^+$ is a candidate key.

→ Decomposition using Functional Dependencies

→ **BCNF : Boyce-Codd Normal Form**

→ a relational schema R is in BCNF w.r.t. a set F of FDs if for all FDs in F^+ satisfy one of these:

① $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$) ; or

② α is a superkey for R

→ **Decomposition**: if in R a non-trivial FD $\alpha \to \beta$ violates BCNF, we decompose R into:
  ① $\alpha \cup \beta$
  ② $R - (\beta - \alpha)$

→ It is not always possible to achieve both BCNF and dependency preservation.

→ Third Normal Form (3NF)
  → R is in 3NF if for all $\alpha \to \beta \in F^+$ at least one of the following holds:
    ① $\alpha \to \beta$ is trivial (i.e., $\beta \subseteq \alpha$); or
    ② α is a superkey for R; or
    ③ each attribute A in $(\beta - \alpha)$ is contained in a contained key for R.

  → is a relaxation of BCNF to preserve dependencies.

→ Algorithms for Functional Dependencies

→ Attribute set Closure
  → Use cases: testing for superkey / candidate key; testing for functional dependencies; and computing closure of F

→ Extraneous Attributes (consider FD $\alpha \to \beta$)
  → Attribute A is **extraneous** in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \to \beta\}) \cup \{(\alpha - A) \to \beta\}$.
  → Attribute A is **extraneous** in β if $A \in \beta$ and $(F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$ logically implies F.

  → To test if A is extraneous in α:
    ① Compute $(\{\alpha\} - A)^+$ w.r.t. F
    ② Check if β is in $(\{\alpha\} - A)^+$
  → To test if A is extraneous in β:
    ① Compute $(\alpha)^+$ w.r.t F', where $F' = (F - \{\alpha \to \beta\}) \cup (\alpha \to \{\beta - A\})$
    ② Check if $(\alpha)^+$ contains A

→ Equivalence of sets of Functional Dependencies
  → F & G are equivalent if $F^+ = G^+$.

→ Canonical Cover

→ F_c is a canonical cover for F if:

① $F^+ = F_c^+$

② no FD in F_c contains extraneous attribute

③ each left side of FD in F_c is unique; i.e., there are no two $\alpha_1 \to \beta_1$ and $\alpha_2 \to \beta_2$ in such that $\alpha_1 \to \alpha_2$

→ is a **minimal set of FDs**

→ Algorithm:

To compute a canonical cover for F:
repeat
  Use the union rule to replace any dependencies in F
    $\alpha_1 \to \beta_1$ and $\alpha_1 \to \beta_2$ with $\alpha_1 \to \beta_1\beta_2$
  Find a functional dependency $\alpha \to \beta$ with an
    extraneous attribute either in α or in β
    /* Note: test for extraneous attributes done using F_c, not F */
  If an extraneous attribute is found, delete it from $\alpha \to \beta$
until F does not change

→ Lossless Join Decomposition

→ Conditions: ① $R_1 \cup R_2 = R$

② $R_1 \cap R_2 \neq \emptyset$

③ $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$

→ Dependency Preservation

→ F_i is the set of dependencies in F^+ that include only attributes in R_i

decomposition is dependency preserving if

$$(F_1 \cup F_2 \cup \cdots \cup F_n)^+ = F^+$$

→ Superkey, candidate key, prime attributes

→ **Superkey** determines every tuple. Any attribute 'A' that does not appear on the R.H.S. of any FD must be in every superkey.

→ **Candidate key** → determines every tuple AND no subset of attributes of candidate key should form superkey.

→ **Prime attributes** → attributes that are a part of at least one candidate key

WEEK 6

→ Desirable Properties of Decomposition

① Lossless Join Decomposition
② Dependency Preserving

→ 3NF relations : free of insertion, update and deletion anomalies.

→ 1NF : First Normal Form

→ iff all domains contain atomic values only
→ example:

Students

SID	Sname	Cname
S1	A	C,C++
S2	B	C++, DB
S3	A	DB

SID : Primary Key

MVA exists ⇒ Not in 1NF

Students

SID	Sname	Cname
S1	A	C
S1	A	C++
S2	B	C++
S2	B	DB
S3	A	DB

SID, Cname : Primary Key

No MVA ⇒ In 1NF

→ cause of redundancy : if $X \rightarrow Y$ is a non-trivial FD over R with X is not a superkey of R, then redundancy b/w X and Y.

→ 2NF : Second Normal Form

→ must be in 1NF and no partial dependency
→ partial dependency :

in $R(X, Y, A)$, If $Y \rightarrow A$, where
Y : proper subset of candidate key, and
A : non-prime attribute. Then, R not in 2NF

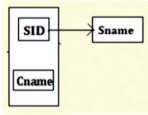
→ example :

STUDENT(Sid, Sname, Cname) (already in 1NF)

Students:

SID	Sname	Cname
S1	A	C
S1	A	C++
S2	B	C++
S2	B	DB
S3	A	DB

(SID, Cname): Primary Key



Functional Dependencies:
{SID, Cname} → Sname
SID → Sname

Partial Dependencies:
SID → Sname (as SID is a Proper Subset of Candidate Key {SID, Cname})

Redundancy?
- Sname

Anomaly?
- Yes

⟹

Key Normalization

R1:

SID	Sname
S1	A
S2	B
S3	A

{SID}: Primary Key

R2:

SID	Cname
S1	C
S1	C++
S2	C++
S2	DB
S3	DB

{SID,Cname}: Primary Key

The above two relations R1 and R2 are
1.Lossless Join
2.2NF
3.Dependency Preserving

→ 3NF : Third Normal Form

→ must be in 2NF and for every FD $X \to A$ either :

① $A \subseteq X$ (i.e., trivial FD) ; or

② X is a superkey ; or

③ A is a prime attribute.

→ **transitive dependency** :

① $A \to B$, and ② not $B \to A$, and ③ $B \to C$, then $A \to C$ is a transitive dependency.

→ **remove transitive dependency** to get 3NF.

→ example :

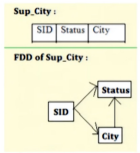
Sup_City(SID, Status, City) (already in 2NF)

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

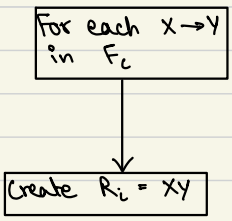
→ Decomposition into 3NF

→ Testing for 3NF

→ no need to check on F^+

→ possible in **polynomial time**.

→ **Algorithm** :



① Compute canonical cover F_c of F.

② Create a relation $R_i = XY$ for each FD $X \to Y$ in F_c.

③ If key K of R does not occur in any relation R_i, create one more relation $R_i = K$.

→ example :

- Relation schema:
  cust_banker_branch = (*customer_id*, *employee_id*, *branch_name*, *type*)
- The functional dependencies for this relation schema are:
  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 the 1st dependency
  - No other attribute is extraneous, so we get F_c =
    customer_id, *employee_id* → *type*
    employee_id → *branch_name*
    customer_id, *branch_name* → *employee_id*

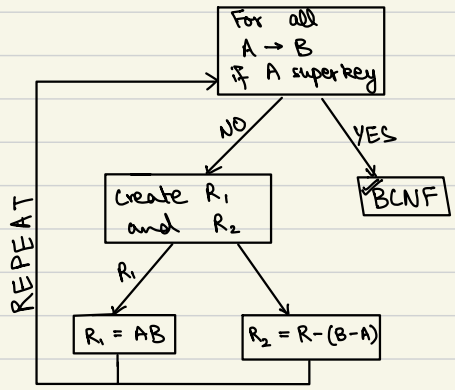
- The **for** loop generates following 3NF schema:
  (*customer_id*, *employee_id*, *type*)
  (*employee_id*, *branch_name*)
  (*customer_id*, *branch_name*, *employee_id*)
  - Observe that (*customer_id*, *employee_id*, *type*) contains a candidate key of the original schema, so no further relation schema needs be added
- At end of for loop, detect and delete schemas, such as (*employee_id*, *branch_name*), which are subsets of other schemas
  - result will not depend on the order in which FDs are considered
- The resultant simplified 3NF schema is:
  (*customer_id*, *employee_id*, *type*)
  (*customer_id*, *branch_name*, *employee_id*)

→ Decomposition to BCNF

→ conditions: For every FD $\alpha \rightarrow \beta$ either:
  ① $\alpha \rightarrow \beta$ is trivial; or
  ② α is a superkey

→ Testing for BCNF
  → no need to check on F^+
  → to test if a decomposed relation is in BCNF, you need F^+, for testing on FDs (for all attributes in R_i) that might not be present in F but can be logically implied.
  → testing for dependency preservation.
    ↳ exponential or polynomial.

→ Algorithm:



① For all FD $A \rightarrow B$ in F^+, check if A is superkey
② If A not superkey,
  create $R_1 = AB$
  create $R_2 = R - (B - A)$
③ Repeat for R_1 and R_2.

3NF	BCNF
① concentrates on Primary key.	concentrates on Candidate key.
② higher redundancy than BCNF.	0% redundancy.
③ FD $X \rightarrow Y$ allowed if X is super key or Y is a part of some key.	FD $X \rightarrow Y$ allowed if X is super key.

→ Multivalued Dependency

→ multi-valued FD notation: $(\rightarrow\!\!\!\!\rightarrow)$ or $(\rightarrow\!\!\!\!\rightarrow)$

→ a table can be in BCNF and still have redundancy because of multi-valued dependencies.

→ **Complementation**

If $X \rightarrow\!\!\!\!\rightarrow Y$, then $X \rightarrow\!\!\!\!\rightarrow (R-(X \cup Y))$

Augmentation

If $X \rightarrow\!\!\!\!\rightarrow Y$ and $W \supseteq Z$, then $WX \rightarrow\!\!\!\!\rightarrow YZ$

Transivity

If $X \rightarrow\!\!\!\!\rightarrow Y$ and $Y \rightarrow\!\!\!\!\rightarrow Z$, then $X \rightarrow\!\!\!\!\rightarrow (Z-Y)$

Replication

If $X \rightarrow Y$, then $X \rightarrow\!\!\!\!\rightarrow Y$ but reverse untrue

Coalescence

If $X \rightarrow\!\!\!\!\rightarrow Y$, and there is W s.t. $W \cap Y = \phi$
$W \rightarrow Z$ and $Y \supseteq Z$, then $X \rightarrow Z$

→ Decomposition to 4NF

→ If for all multi-valued dependencies in D^+ of form $\alpha \rightarrow\!\!\!\!\rightarrow \beta$ either:

① $\alpha \rightarrow\!\!\!\!\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)

② α is a superkey

→ If R is in 4NF then it is in BCNF.

→ Temporal Databases

→ time-varying data

→ have an associated time interval during which it is valid

→ **Snapshot**: value of data at a particular point in time

→ **Timestamp Model**

→ Two aspects of time:

Historical ① **Valid time**: time period which an attribute is valid in the real world.

Rollback ② **Transaction time**: time period for which the attribute is stored in the database.

→ Uni-temporal relation → either of the two aspects of time.
Bi-temporal " → both aspects

→ Problem solving on normalisation

Q. Let R(A,B,C,D) where all attributes have atomic values
FD = {B → C, D → A}. Find the highest normal form R is in
A.

Candidate keys = {BD}
All atomic values so 1NF ✓
B → C and B is proper subset of Cand key and C is non-prime
2NF ✗

Q. R(A,B,C,D,E,F). FD = {BCD → E, BCE → F, DE → A, DEA → C, CA → E}
A

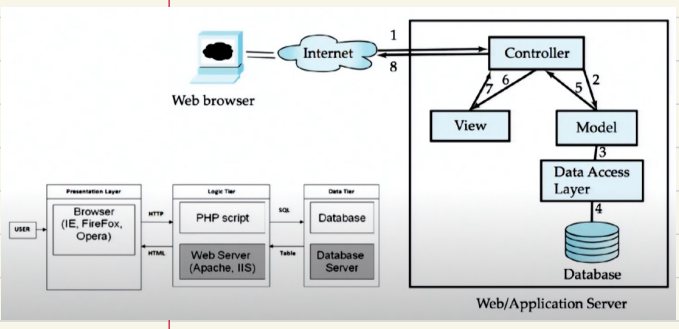
Candidate keys = {BCD, BDE}
2NF ✗ because of BDE and DE → A

WEEK 7

→ Characteristics of Application Programs

→ Applications are split into:

① Frontend Layer (presentation layer)
↳ interacts with the user
↳ example: choose item, checkout, etc.
→ MVC architecture
→ model: business logic
→ view: presentation of data
→ controller: receive actions, return view



② Middle Layer (application / business logic)
↳ functionality of application
↳ login authentication, search logic, etc.
→ high level view of data
→ hides details of data storage schema

③ Backend Layer (data access)
↳ manages persistent data
→ interfaces b/w business logic layer and underlying database.

→ Business Logic Layer
  ↳ abstraction of entities
  ↳ gives you the object model
    ↳ so you can use Java or C++
  ↳ supports workflows

  ↳ Object - Relation Mapping
    → looks at every entity as an object
    → allows you to create an object of the relational model in the database so that you can use object-oriented programming

→ Tiered Architecture

→ 1-tier : all components for an application on a single server

→ 2-tier : client-server architecture
  → doesn't give separation b/w front-end and back-end

→ 3-tier : Presentation, Logic, and Data Acess
  → most widely used to design a DBMS

→ URI < URL → locator / address
      < URN → name

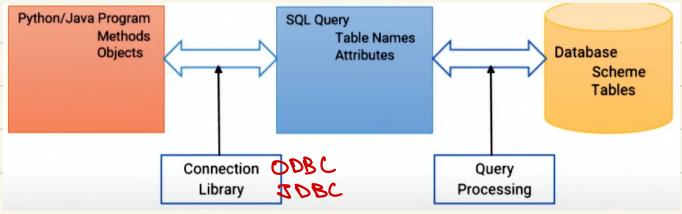
→ Sessions and cookies
  → HTTP protocol is sessionless
    ↳ i.e., once a server replies to a request, server closes the connection.
  → Cookie : small piece of information
    ↳ text containing identifying information.
    ↳ can be stored permanently or limited time.

→ Java Server Pages
  → similar concept as servlet
  → compiled into Java + Servlets
  → much easier to write HTML embedding
  → platform independent
  → server-side
  → can contain dynamic information unlike HTML

→ Applications use API to interact with a database server
  → connect with server; send SQL commands; fetch tuples of result one-by-one

  → Frameworks
    ① Connectionist
      → ODBC works with C, C++, Python
      → JDBC works with Java
      → SQL program is created, executed through a set of APIs.
    ② Embedding
      → put SQL commands in your native language

→ Connectionist

→ 

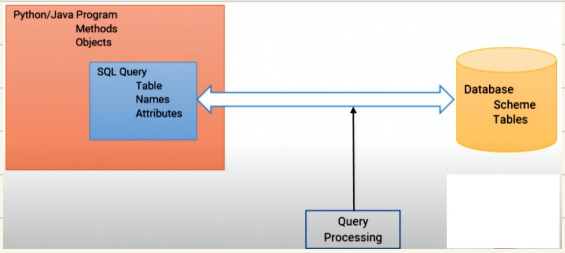
→ ODBC
  → Open Database Connectivity: standard API for accessing DBMS
  → will work on all operating systems and database systems
  → python example using pyodbc library →

```
import pyodbc

conn = pyodbc.connect('DSN=SQLS;UID=test01;PWD=test01')
cursor=conn.cursor()
cursor.execute("create table rvtest (col1 int, col2 float,
col3 varchar(10))")
cursor.execute("insert into rvtest values(1, 10.0,
\"ABC\")")
cursor.execute("select * from rvtest")

while True:
      row=cursor.fetchone()
      if not row:
            break
      print(row)


cursor.execute("delete from rvtest")
cursor.execute("insert into rvtest values (?, ?, ?)", 2,
20.0, 'XYZ')
cursor.execute("select * from rvtest")

while True:
      row=cursor.fetchone()
      if not row:
            break
      print(row)
```

→ JDBC
  → Java Database Connectivity: API for the language Java, which defines how a client may access a database

→ Bridge Drivers
  → bridges b/w JDBC and ODBC
  → translates source function calls into target function calls

→ Embedding

→ 

→ Uses a tag and then puts the SQL statement inside it.

EXEC SQL <embedded SQL statement>

→ EXEC SQL
declare c cursor for (output available in this cursor)
select ID, name
from student
where tot_cred > :credit_amount (variable from the host language)
END_EXEC
EXEC SQL open c ;
EXEC SQL fetch c into :si, :sn END_EXEC
EXEC SQL close c ;

→ Postgre SQL from Python

→ steps to access the database:

① Create connection
② Create cursor
③ Execute query
④ Commit | rollback
⑤ Close the cursor
⑥ Close the connection

→ example:

```python
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()
connectDb('mydb', 'myuser', 'mypass', '127.0.0.1', "5432") # function call
```

→ create a new table

```python
import psycopg2

def createTable():

    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
        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()
        print("Table created successfully")
        cur.close()

    except (Exception, psycopg2.DatabaseError) as error:
        print(error)

    finally:
        if conn is not None:
            conn.close()

createTable() # function call
```

→ Flask Web App Framework
  → lightweight WSGI
  → app.route ("<URL>")
    ↳ URL binding with the function

→ Rapid Application Development
  → To speed up app development
    → function library to generate user-interface elements
    → drag-and-drop features in IDE to create UI elements (VS code)
    → automatically generate code for UI from a declarative spec
  → Phases:
    ① Business modeling
    ② Data modeling
    ③ Process modeling
    ④ Testing & turnover

  → Frameworks
    → Java Server Faces
      → a set of APIs for representing UI components

    → Ruby on Rails
      → easy creation of simple CRUD interfaces

  → Platforms and tools
    G Suite, Google App Engine, Microsoft Azure, AWS Elastic

→ Application Performance
  → caching techniques to reduce cost of serving pages.
  → can be at server site or at the client's network.

→ Application Security
  → SQL injection use prepared statements with user inputs as parameters
  → Never store passwords in clear texts, encrypt it very well
  → restrict access to database by source IP address
  → Authentication
    → password plus OTP
    → face recognition, fingerprint, etc.

  → Audit trail at the database and the application level
    → are used after-the-fact

→ Types of Mobile Apps:
  ① Native: native language of a platform.
  ② Web: runs completely inside of a web browser
  ③ Hybrid: combines attributes of both.

X X X

WEEK 8

→ **Algorithm** : finite sequence of well-defined, computer-implementable instructions.
  ↳ must terminate

→ **Program** : collection of instructions
  ↳ may or may not terminate

→ **Physical Storage Media**
  → speed
  → cost per unit of data
  → reliability

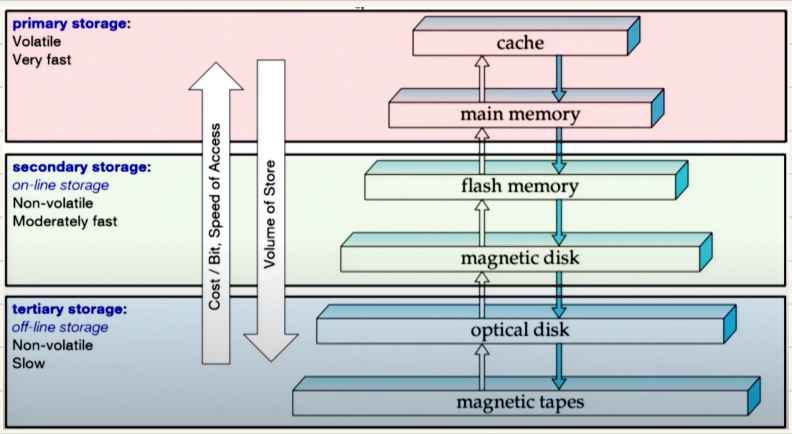
  → **Volatile storage** : loses contents when power is switched off
    → cache : very fast and most costly storage
    → main memory : fast access ; very small

  → **Flash memory** : data survives power failure
    → reads as fast as main memory

  → **Magnetic Disk** : primary medium for long-term storage
    → direct-access : possible to read data on disk in any order, unlike magnetic tape
    → survives power failure

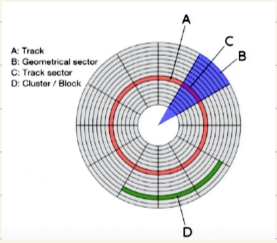
  → **Optical Storage** : write once and read many times (archival data)

  → **Tape storage** : non-volatile ; for backup and archival
    → sequential access
    → very high capacity



→ Magnetic Disk

→ Mechanism



→ surface divided into circular tracks

→ each track divided into sectors

→ Disk controller : interfaces b/w computer system and disk drive hardware

↳ computes and attaches checksums to each sector to verify that correct read back

→ Access time : Seek time and Rotational Latency

↳ Seek time : time to reposition the arm over the correct track

↳ Rotational Latency : to get to that particular sector

→ Data-transfer rate

↳ typically 25 to 100 MB per second

→ Mean time to failure (MTTF)

↳ typically 3-5 years

→ Magnetic Tapes

→ large volumes of data but lesser than disks

→ very cheap

→ limited to sequential access

→ Cloud Storage

→

Parameters	Cloud Storage	Traditional Storage
Cost	Cloud storage is cheaper per GB than using external drives.	The hardware and infrastructure costs are high and adding on more space and upgrading only adds extra costs.
Reliability	Cloud storage is highly reliable as it takes less time to get under functioning	Traditional storage requires high initial effort and is less reliable.
File Sharing	Cloud storage supports file sharing dynamically as it can be shared anywhere with network access	Traditional storage requires physical drives to share data and a network is to be established between both
Accessibility	Cloud storage gives you access to your files from anywhere	Restricted to local access
Backup/ Recovery	Very safe from on site disaster. In case of a hard drive failure or other hardware malfunction, you can access your files on the cloud, which acts as a backup solution for your local storage on physical drives	Data that is stored locally is much more susceptible to unexpected events and local storage and local backups could be easily lost

→ Future of Storage

→ DNA digital storage

→ Quantum memory

→ File Organisation

→ Database = collection of files
File = sequence of records
Record = sequence of fields

→ database file is partitioned into fixed-length storage units called **blocks**

→ Deletion of a record i : Alternatives
① move records $i+1, \cdots, n$ to $i, \cdots, n-1$
② move record n to i
③ do not move records but link all free records on a **free list**.

→ Variable-length records
→ storage of multiple record types in a file
→ record types that allow variable length for fields
→ slotted page header contains:
→ number of record entries
→ end of free space in a block
→ location and size of each record

→ Organisation of records in files

→ Heap : record placed anywhere in the file where there is space
Sequential : in sequential order based on value of search key of each record
Hashing : hash function computed on some attribute of each record

→ Sequential file organisation
↳ deletion : use pointer chains
insertion : locate position ; if no free space insert into **overflow block**
↳ need to reorganise to restore the order

→ Multitable clustering
↳ Store several relations in one file using a **multitable clustering** file organiza

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

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

multitable clustering of department and instructor

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

group records of instructors from the same department

→ good for queries involving department ⋈ instructor

→ Data Dictionary Storage

→ data dictionary stores **metadata**

↳ information about relations

↳ statistical and descriptive data

↳ physical file organisation information

→ Storage Access

→ minimise number of block transfers b/w disk and memory

→ **buffer** : portion of main memory available to store copies of disk blocks

→ **buffer manager** : subsystem responsible for allocating buffer space

↳ programs call on buffer manager when they need a block from disk

↳ **LRU strategy** : Least Recently Used → use past patterns of block references as a predictor of future references

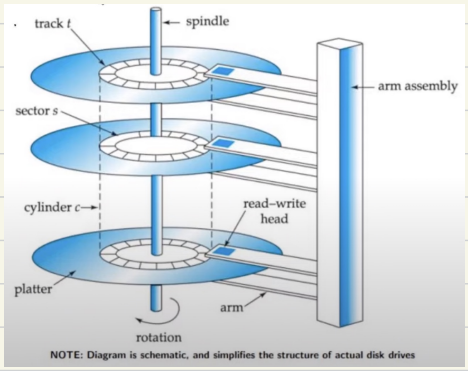
↳ **Pinned block** : memory block that is not allowed to be written back to disk

↳ **toss-immediate strategy** : free the space occupied by a block as soon as the last tuple has been processed.

↳ **MRU strategy** : Most Recently Used → pin the block currently being used; unpin the block once processed.

→ Tutorial

→



→ sector size : 512 byte

→ number of cylinders = number of tracks per surface

WEEK 9

→ Indexing

→ use a separate table for 'indexed' attribute.
The index table is sorted on the attribute and also has an additional attribute 'pointer' that points to the tuples in original data.

→ kinds of indices
↳ ordered indices
↳ hash indices

→ index evaluation: access types supported; access time; insertion time; deletion time; space overhead

→ Ordered Indices

→ entries stored in sorted order on the search key (indexed attribute) value.

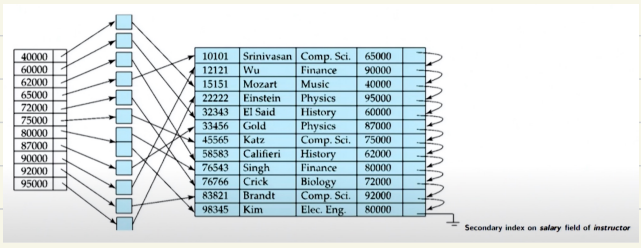
→ primary index (a.k.a. clustering index)
↳ in a sequentially ordered file, the index whose search key specifies the sequential order of the file
↳ usually (but not always) primary key.

→ secondary index (a.k.a. non-clustering index)
↳ index whose search key specifies a diff. order from the sequential order of the file.

→ dense index
↳ index record appears for every search-key value in the file.
↳ might become very large

→ sparse index
↳ index records for only some search-key values
↳ to find record with search-key value K, find index record with largest search-key value < K
↳ first record of every block as index entry.

→ if secondary index has repeated values, maintain only distinct
↳ index record points to a bucket that contains pointers
↳ has to be dense



→ multilevel index

↳ if primary index does not fit in memory

↳ solution: treat primary index file as sequential file and construct a sparse index on it.

outer index: sparse index on primary index

inner index: primary index file

→ Deletion in index

↳ dense: delete similar search-key in index table

↳ sparse:

→ if entry of the search-key exists in the index table, delete it and replace with the next search-key value in the file (in search-key order)

→ Insertion in index

↳ dense: if search-key value doesn't appear in index, insert it

↳ sparse:

→ if a new block is created, the first search-key in the new block is inserted into the index

→ Balanced Binary Search Trees

→ BST is balanced if height ~ log(n)

→ Balancing strategies:

→ AVL tree → Skip List

→ Randomised BST → Splay

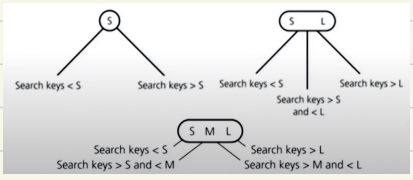
→ These strategies work well for in-memory operations, small volume of data but have complex operations and do not scale for external data

→ 2-3-4 Tree

→ guarantee that all leaves are at same depth (bottom level)

↳ height of all leaf nodes are same; h ~ O(log n)

→ use three kinds of nodes:



→ Insertion: search to find expected location:

→ if 2 node: change to 3 node and insert

→ if 3 node: change to 4 node and insert

→ if 4 node: split the node by moving the middle item into parent node, and insert

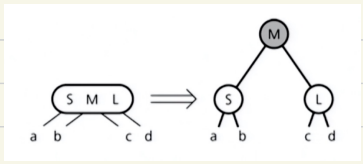
→ node splitting if:

↳ 4-node is the root ; or

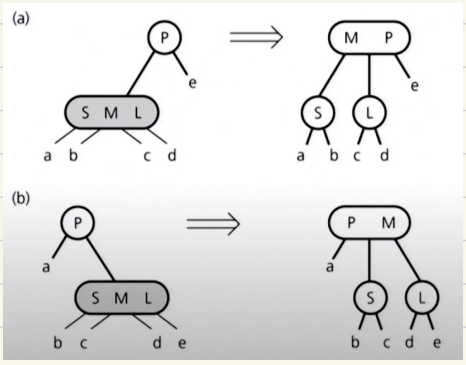
↳ 4-node has a 2-node parent ; or

↳ 4-node has a 3-node parent

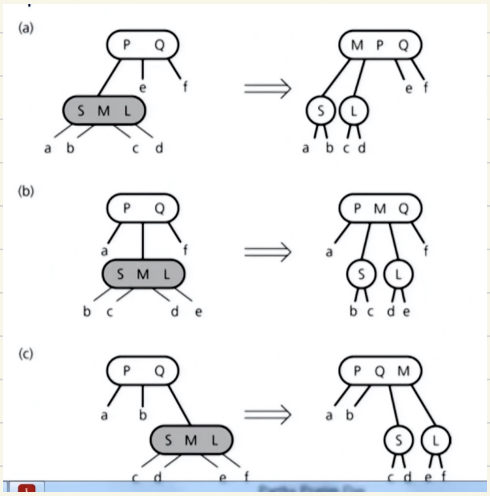
→ splitting at root



→ splitting with 2-node parent



→ splitting with 3-node parent



→ height only changes when the split happens at the root but the height for all leaf nodes is still equal.

→ strategies:

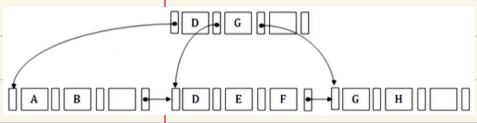
early → split a 4-node as soon as you come across it in traversal ; ensures that the path doesn't have multiple 4-nodes.

late → split a 4-node only when you need to insert an item in it ; could lead to O(h) splits

↳ both strategies have the same complexity.

→ B⁺ Tree Index Files

→ B⁺ Tree : balanced binary search tree



↳ follows multi-level index like 2-3-4 tree

↳ leaf nodes denote actual data pointers

↳ leaf nodes are linked using a link list

↳ internal nodes

→ at least $n/2$ child pointers except root

→ at most n pointers

↳ leaf nodes

→ at least $n/2$ record pointers and $n/2$ key values

→ at most n " " " n " "

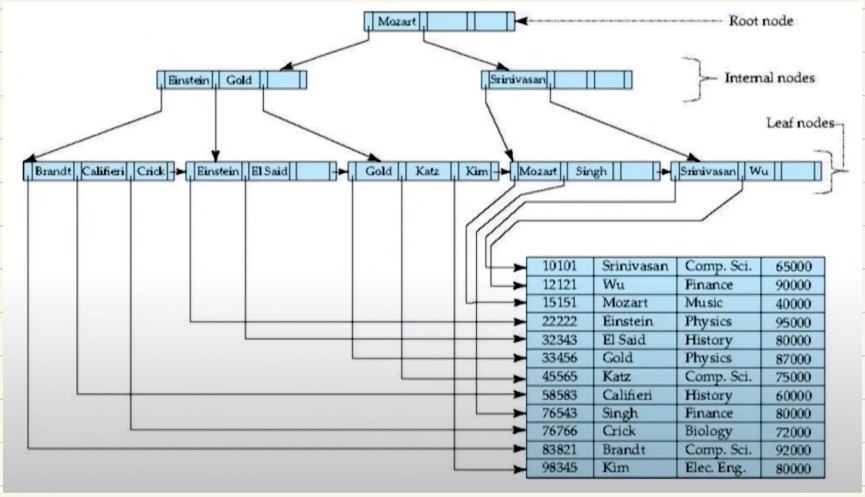
↳ insertion

→ if the target location is full, split the node following the process as in 2-3-4 tree.

↳ re-organises itself, as opposed to Index Sequential file

↳ disadvantage: extra insertion and deletion overhead

↳ example:



↳ non-leaf levels form a hierarchy of sparse indices

↳ since every node must have $n/2$ values, to search for a key value, it takes $O(\log_{(n/2)} K)$, where K = total # of keys.

↳ can also be used for database files.

→ B- Tree

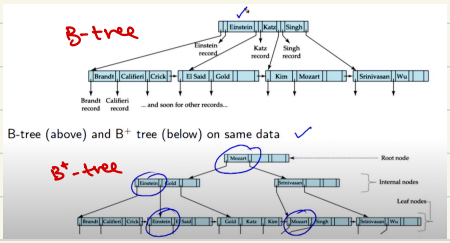
→ similar to B⁺ tree but allows search-key values to appear only once.

→ uses less tree nodes

→ sometimes possible to find before reaching leaf nodes

→ non-leaf nodes are larger; height increases

→ insertion and deletion becomes more complicated.



→ Static Hashing

→ Hash function

↳ maps data of arbitrary size (from domain D) to fixed-size values (say, integers from 0 to N>0).

$h : D \rightarrow [0, .., N]$

→ bucket : unit of storage containing one or more records

→ records with diff. search-key values may be mapped to the same bucket so the entire bucket has to be searched sequentially

→ ideal hash function is uniform : each bucket is assigned same number of search-key values

→ ideal hash function is random

→ bucket overflows reasons:

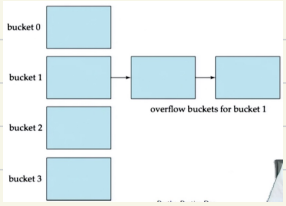
→ insufficient buckets

→ skew is distribution of records

handling:

→ overflow buckets

↳ closed hashing



→ problematic because of fixed number of buckets.

→ Dynamic Hashing

→ extendable hashing ; hash function generates values over a large range → typically b-bit integers with b = 32

→ use only a prefix of hash function to index into a table of bucket addresses.

↳ Let length of prefix be i bits, $0 \leq i \leq 32$

→ bucket address table size = $(2)^i$

→ value of i grows and shrinks with the database

↳ example with b = 3

① 0 0 0 ⑤ 1 0 0

② 0 0 1 ⑥ 1 0 1

③ 0 1 0 ⑦ 1 1 0

④ 0 1 1 ⑧ 1 1 1

with i = 1, bucket 0 = {①, ②, ③, ④}

bucket 1 = {⑤, ⑥, ⑦, ⑧}

with i = 2, bucket 0 = {①, ②} 2 = {⑤, ⑥}

bucket 1 = {③, ④} 3 = {⑦, ⑧}

→ insertion:
  → search: get to index, get the hash, take the prefix, go to the bucket address table, then go to the bucket.
  → if target bucket is full: options:
    ① allocate a new bucket
    ② if you can't create anymore buckets, create an overflow bucket OR use more bits

→ Bitmap Index
  → for attributes that have a limited number of possible values, like gender
  → has as many bits as records
  → in a bitmap for value v, the bit of a record is 1 if the record has the value v for the attribute, and is 0 otherwise

→ Index Definition in SQL
  → create index <index-name> on <relation-name> (<attribute-list>)
  → Tablespace → can specify which tablespace to go to

  → STORAGE → how to store a database object
    ↳ INITIAL size of the 1st extent of object
    ↳ NEXT " " " 2nd " " "
    ↳ PCTINCREASE how much %age later extents grow
    ↳ PCTFREE % of each data block be free for updates

  → Multiple keys
    ↳ multiple indices; composite index
    ↳ composite search key (order is important)
      → lexographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
        → $a_1 < b_1$, or
        → $a_1 = b_1$ and $a_2 < b_2$

  → To create an index
    ↳ you must own, or have the INDEX object privilege

→ Guidelines for Indexing

→ collect statistics about data to learn the patterns and adjust indexes

→ Ground Rules:

① Indexes lead to Access-Update tradeoff
② Index the correct tables
③ Index the correct columns
④ Limit the number of indexes for each table
⑤ Choose the order of the columns in composite indexes
⑥ Gather statistics about usage
⑦ Drop indexes no longer required

WEEK 10

→ Transactions

→ unit of program execution that accesses and possibly updates, various data items

→ ACID properties

→ Atomicity (All or nothing transactions)

↳ ensure updates of partial transaction are not in database

→ Consistency (Guarantees committed transaction state)

↳ explicitly specified integrity constraints (primary / foreign keys)
↳ implicit integrity constraints

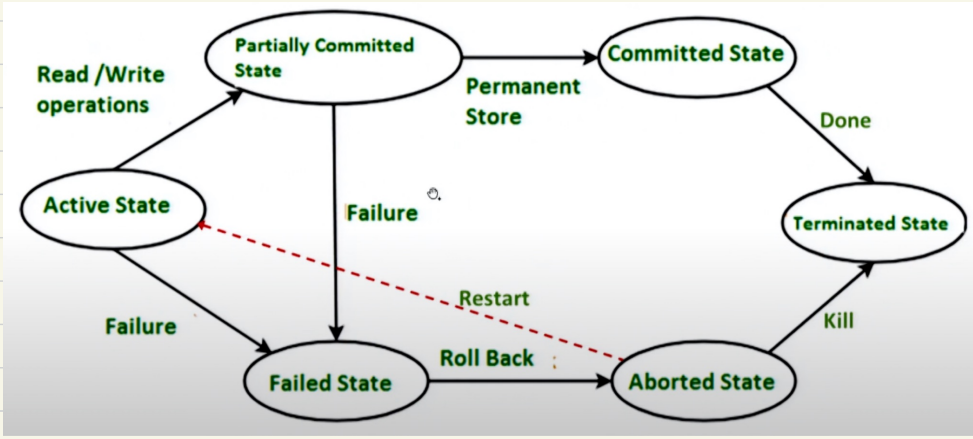
→ Isolation (Transactions are indepent)

↳ if multiple transactions, no inconsistent updates are made together
↳ run transactions serially

→ Durability (committed never lost)

↳ once a transaction is committed, it will remain committed

→ States ⇒



→ Concurrent Executions

- → multiple transactions allowed to run concurrently
  - ↳ increased disk utilization
  - ↳ reduced average response time
- → **Schedule**: sequence of instructions
  - ↳ must consist all transactions
  - ↳ must preserve order of instructions within transaction
- → successful (failed) transaction will have commit (abort) at the end.
- → in a schedule, the database might be incosistent state at one of the commits but that is acceptable as long as the database is in a **consistent state after the last commit**.
- → all schedules may not satisfy ACID

→ Serializability

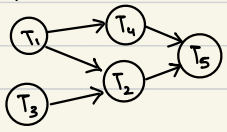
- → assumption: each transaction preserves db consistency
- → schedule is serializable if **equivalent to a serial schedule**
  - ↳ conflict serializability
  - ↳ view serializability

- → conflicting instructions
  - ↳ Two instructions I1 and I2 accessing the same item if I1 = 'write' or I2 = 'write':
    - I1 and I2 are conflicting
    - → if they're consecutive no conflict

→ Conflict Serializability

- → Transform schedule S into S' by swapping non-conflicting instruction. S and S' are conflict equivalent
- → Schedule S is conflict serializable if it is **conflict equivalent to a serial schedule**.
- → If a schedule is serializable, it doesn't necessarily mean that it is conflict serializable.
  - ↳ ex. $w_1(A), w_2(A), w_2(B), w_1(B), w_3(B)$
    serial: $w_1(A), w_1(B), w_2(A), w_2(B), w_3(B)$

- → Precedence Graph
  - ↳ nodes: transactions
    - ↳ if two transactions conflict then they're connected
  - ↳ a schedule is **conflict serializable iff the precendence graph is acyclic**.
    - ↳ serializability can be obtained by topological sorting
  - ↳ example: $w_1(A), r_2(A), w_1(B), w_3(C), r_2(C), r_4(B), w_2(D), w_4(E), r_5(D), w_5(E)$



T_1, T_3, T_4, T_2, T_5

→ Recovery

→ if transaction T_j reads an item previously written by T_i, then the commit operation of T_i must appear before commit of T_j
↳ then this schedule is recoverable in case of a system crash

→ cascading rollback
↳ example →

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)

— schedule is recoverable because no transaction commits

If T_{10} fails, ← abort
T_{11} and T_{12} must also be rolled back

→ cascadeless schedule
↳ for each T_i and T_j, T_j reads an item written by T_i, the commit of T_i appears before read operation of T_j
↳ every cascadeless is also recoverable

Example: Irrecoverable Schedule

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A – 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
		Commit;		
Failure Point				
Commit;				

Example: Recoverable Schedule with Cascading Rollback

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A – 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
Failure Point				
Commit;				
		Commit;		

Example: Recoverable Schedule without Cascading Rollback

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A – 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
Commit;				
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
		Commit;		

→ Transaction Definition is SQL

→ transaction end by: Commit work or by Rollback work

→ TCL Transaction Control Language

COMMIT	→ save changes
ROLLBACK	→ roll back change
SAVEPOINT	→ points within groups of transactions in to rollback
SET TRANSACTION	→ names a transaction

↳ are only used with DML commands: INSERT, UPDATE, DELETE

→ View Serializability

→ weaker than conflict serializability

→ S and S' schedules are view equivalent if:

① Initial Read

↳ If T_i in S reads initial value of Q, then is S' also T_i must read initial value of Q.

② Write - Read Pair

↳ If T_i is S executes read(Q) which reads the value written by T_j, then in S' this order of write - read must be maintained

③ Final Write

↳ The transaction that performs the final write(Q) operation in S must also be the last one in S'.

→ view serializable example: (but not conflict serializable)

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

T_{28} and T_{29} write(Q) are **blind writes**

→ Concurrency Control

→ develop concurrency control protocols that will assure serializability

→ one way to ensure isolation: require that data items be accessed in a mutually exclusive manner.

→ allow a transaction to access a data item only if it is holding a **lock** on that data item.

→ Lock - Based Protocol

→ data items locked in two modes:

① exclusive (X) mode

↳ both read and write

↳ using lock-X instruction

② shared (S) mode

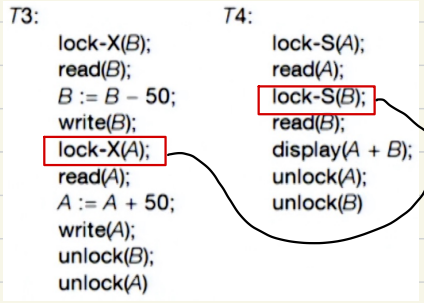
↳ only read

↳ using lock-S instruction

→ unlock data item Q by unlock-S instruction

→ Deadlock when neither transaction can ever proceed because of a 'loop' in locking of data items.

↳ example:



↳ the system must roll back one of the transactions.

→ Two-Phase Locking Protocol

↳ Phase 1: Growing Phase

↳ transaction obtains but not release locks

Phase 2: Shrinking Phase

↳ release locks but not obtain

↳ assures serializability ; in the order of lock points

→ Lock Conversions

↳ Phase 1

↳ can lock-S on item

" lock-X on item

" convert lock-S to lock-X

Phase 2

↳ can release lock-S

" " lock-X

" convert lock-X to lock-S

↳ doesn't ensure freedom from deadlock

→ Starvation (Livelock)

↳ when a transaction is trying to get an X-lock while a sequence of other transactions request and are granted an S-lock on the item.

→ Strict Two-phase protocol

↳ transaction must hold all its X-locks till it commits/aborts.

↳ Rigorous → all locks held until commit/abort

→ Deadlock Handling

→ strategies:

→ each transaction locks all its data items before execution

→ partial ordering

→ Transaction timestamp · created by DBMS to identify relative starting time

① **Wait-die** scheme : non-preemptive

↳ older transactions wait for younger ones

↳ ex. T_5 requests data held by T_{10} will wait

T_{15} requests " " " " " be killed

② **Wound-wait** scheme : preemptive

↳ older transactions forces rollback of younger ones

↳ ex. T_5 requests data held by T_{10} , T_{10} will be wounded

T_{15} requests " " " " , T_{15} will wait

→ Deadlock detection

↳ graph with transactions as nodes.

If T_i is waiting for T_j , then there is an edge from T_i to T_j

↳ if graph has cycle , it has a deadlock .

→ Deadlock Recovery

↳ most effective to rollback to the lockpoint that has caused the deadlock .

↳ include number of rollbacks in the cost factor to avoid starvation .

→ Timestamp Based Protocols

→ maintains for each data Q :

→ W-timestamp (Q) · largest successful write(Q)

→ R-timestamp (Q) : " " read (Q)

→ T_i issues a **read (Q)** :

if $TS(T_i) \leq$ W-timestamp(Q) : read rejected and T_i rolled back

if $TS(T_i) \geq$ W-timestamp(Q) : read accepted and R-timestamp(Q) update

→ T_i issues a **write (Q)** :

if $TS(T_i) <$ R-timestamp(Q) : write rejected ; T_i rolled back

if $TS(T_i) <$ W-timestamp(Q) : " " " " "

else allow write operation and update W-timestamp(Q)

→ no cycles in precedence graph

→ ensures deadlock-free

→ may not be cascadeless or recoverable

WEEK 11

→ Backup and Recovery

- → Backup: representative copy of data
  - ↳ Physical Backup
    - ↳ copy of physical database files
    - ↳ e.g. data, control files, log files
  - ↳ Logical Backup
    - ↳ copy of logical data extracted from db
    - ↳ e.g. tables, procedures, views, functions
- → uses of backup
  - ↳ disaster recovery
  - ↳ client side changes
  - ↳ auditing
  - ↳ downtime
- → Types of backup data
  - ↳ Business data: personal info of clients, employees
  - ↳ System data: environment / configuration of system
  - ↳ Media: photos, videos, graphics

→ Backup Strategies

- → Full Backup
  - ↳ backs up everything
  - ↳ must be done at least once before any other type of backup
  - ↳ frequency depends on application
- → Incremental Backup
  - ↳ backup only files that have changed since the last backup
  - ↳ cannot be done without full backups
- → Differential Backup
  - ↳ backup only files that have changed since the last full backup

→ Hot Backup

→ preferable wherever possible
→ refer to keeping a database up and running while the backup is performed concurrently
→ database is always available
→ detecting error can be difficult

→ mainly used for Transaction Log Backup

→ Failure Classification

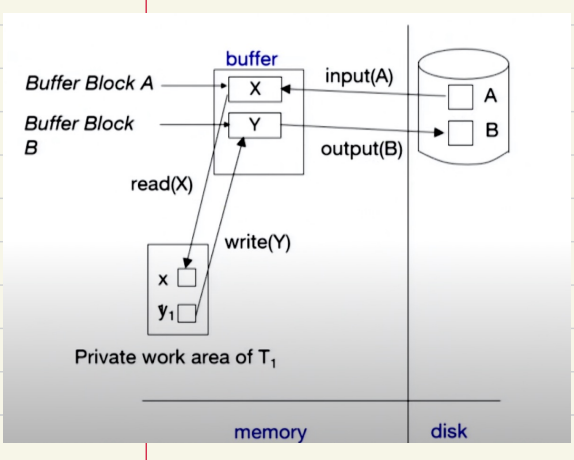
→ Transactions have ACID properties
→ Transaction failure
→ logical errors
→ system errors
→ System crash
→ Disk crash

→ Recovery Algorithms
↳ actions taken during transaction processing to ensure enough information exists to recover from failures
↳ actions taken after a failure to recover db contents to state that ensures atomicity, consistency, durability

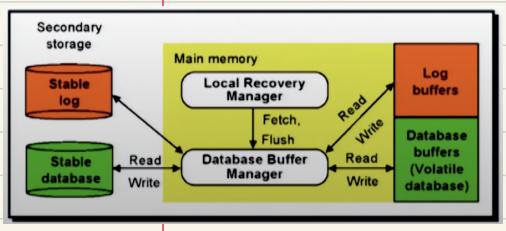
→ Storage Structure

→ Volatile : does not survive system crash
↳ main memory, cache memory
→ non volatile : survives system crash
↳ disk, tape, flash memory
→ Stable storage : mythical form that survives all failures
↳ multiple copies on distinct non-volatile media

→ Stable storage implementation
↳ assuming two copies of each block
↳ the output is complete only after 2nd write completes
→ Failure during data transfer.
↳ find inconsistent blocks
↳ record in-progress disk writes and use this information during recovery to find inconsistent blocks
↳ if either copy has an error, overwrite it by the other
↳ if both have no error, overwrite 2nd block by the 1st block



→ Log-Based Recovery



→ Log is created for every transaction

→ when a transaction T_i starts, it registers itself by writing a record $< T_i$ **start** $>$ to the log.

↳ when T_i executes **write(X)**, log record $< T_i, X, V_1, V_2 >$ is written, where V_1 = old value and V_2 = new value

↳ finally T_i finishes last statement, $< T_i$ **commit** $>$ recorded

→ Database Modification Schemes

↳ Immediate Modification allows updates of uncommited transaction to be made to the buffer, or the disk itself

↳ Deferred Modification performs updates to buffer | disk only at the time of transaction commit.

→ Transaction is said to have committed when its commit log record is output to stable storage.

→ Undo of a log record $< T_i, X, V_1, V_2 >$ writes the old value V_1 to X

Redo " " " " " " " " new " V_2 to X

→ Undo is used for rollback during normal operation

Undo and Redo are used during recovery from failure

→ Checkpoints

↳ to streamline recovery

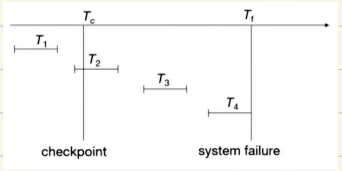
↳ All updates are stopped while doing checkpointing

① output all log records from main memory to storage

② output all modified buffer blocks to disk

③ write log record $<$ **checkpoint** $L >$ onto storage where L is a list of all transactions active at the time of checkpoint

↳ example:



T_1 can be ignored.

T_2 and T_3 redone.

T_4 undone

commits before checkpoint: ignored

commits after checkpoint: redone

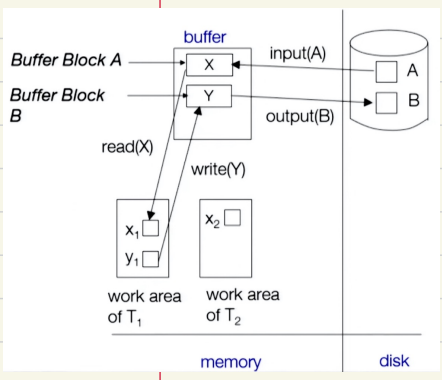
uncommitted transactions: undone

→ Transactional Logging

→ Hot backup
↳ take a backup while doing transactions

→ **Recover**: retrieve from the backup media, db files, and transactions log
Restore: re-apply db consistency based on the transaction log

→ Recovery Algorithm

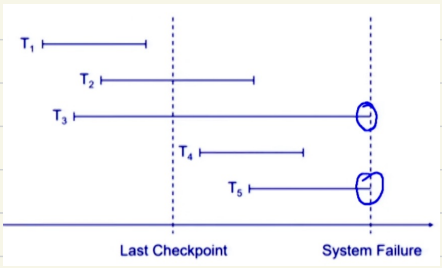


→ assume if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted.

→ Logging (normal operation):
$< T_i$ **start** $>$
$< T_i, X, V_1, V_2 >$
$< T_i$ **commit** $>$

→ Transaction rollback (normal operation):
↳ scan the log backwards and undo each log record
→ for $< T_i, X, V_1, V_2 >$, undo X to V_1 and write a log record $< T_i, X, V_1 >$ (Compensation Log Record)

→ example:



- T_1 ignored
- T_2 and T_4 redone
- T_3 and T_5 undone and restarted

→ **Redo phase**:
① Find last $<$ **checkpoint** L $>$ record, and set undo-list L
② Scan forward from $<$ **checkpoint** L $>$ record
→ whenever record $< T_i, X, V_1, V_2 >$ found, redo it
→ " " $< T_i$ **start** $>$ found, add T_i to undo-list
→ " " $< T_i$ **commit** $>$ or $< T_i$ **abort** $>$ found, remove T_i from undo-list
→ At the end of this phase, undo-list will have all the transactions that need to be undone.
→ Then do the **undo phase** (reverse each action).

→ Recovery with Early Lock Release

→ any index used in processing a transaction, such as a B^+-tree, can be treated as normal data.

→ B^+-tree concurrency algorithm allows locks to be released early

↳ because of early release it is possible:

→ T_1 updates a value in B^+ tree by inserting entry (V_1, R_1)

→ T_2 inserts (V_2, R_2) in the same node, moving the entry (V_1, R_1) before completion of T_1

↳ we cannot undo T_1 since that would also undo the insert done by T_2

↳ the only way to undo the effect of insertion (V_1, R_1) is to execute a deletion.

→ Logical undo

→ Operation Logging: Process

↳ Operation starts → log $< T_i, O_j,$ **operation-begin** >

↳ while executing operation, log records in the normal fashion

↳ Operation completes → $< T_i, O_j,$ **operation-end**, $U >$

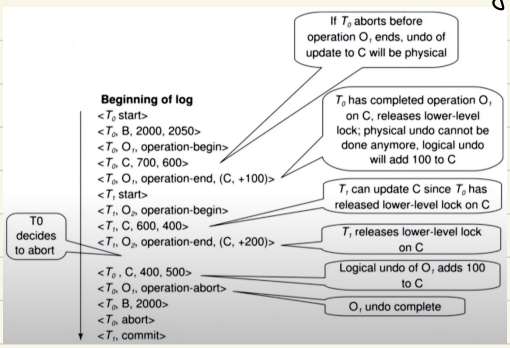
contains information needed to perform a logical undo

↳ example:

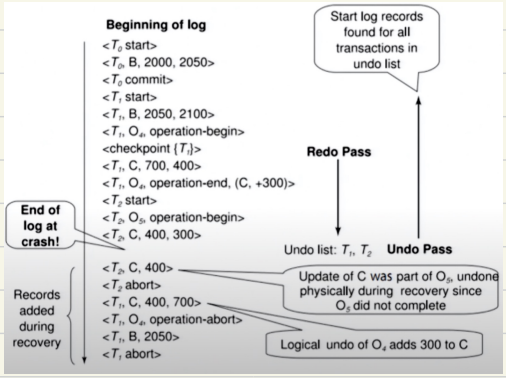
- Insert of (key, record-id) pair (K5, RID7) into index I9

<T1, O1, operation-begin>
....
<T1, X, 10, K5>
<T1, Y, 45, RID7>
} Physical redo of steps in insert
<T1, O1, operation-end, (delete I9, K5, RID7)>

→ example of Transaction Rollback with Logical Undo

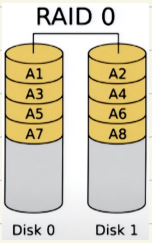


→ example of Failure Recovery with Logical Undo

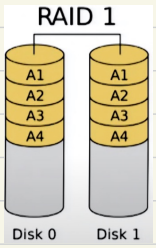


→ Plan for Backup and Recovery
- → Data Importance
- → Frequency of change
- → Speed of backup or recovery
- → Equipment
- → Employees
- → Storing

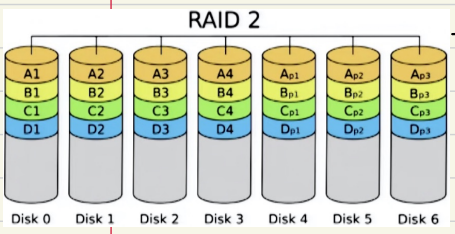
→ RAID: Redundant Array of Independent Disks
- → disk organisation techniques that manage a large number of disks providing a view of a single disk with:
  - ↳ high capacity and high speed
  - ↳ high reliability
- → Mirroring
  - ↳ duplicate every disk
  - ↳ every write done on both disks
  - ↳ read from either
  - ↳ data loss would occur only if one disk fails and its mirror disk also fails before the system is repaired
- → Bit-level Striping
  - ↳ splits bits of each byte across multiple desks
  - ↳ seek / access time worse than for a single disk
- → Byte-level Striping
  - ↳ each file split into parts one byte in size
- → Block-level striping
  - ↳ with n disks, block i of a file goes to disk $(i \bmod n)+1$
  - ↳ requests for diff. block can run in parallel if they reside on different disks
- → Bit-Interleaved Parity
  - ↳ single parity bit enough for error detection and correction
- → Block-Interleaved Parity
  - ↳ uses block-level striping and keeps a parity block on a separate disk for corresponding blocks from n other disks
- → Standard RAID levels
  - ↳ RAID 0 : striping ; RAID 1 : mirroring ; RAID 5 : distributed parity; RAID 6 : dual parity
  - ↳ can combine (or nest) levels : RAID 01 : mirroring stripe sets RAID 10 : stripping of mirrors
- → RAID only a building block of a data loss prevention and recovery scheme but cannot replace a backup plan



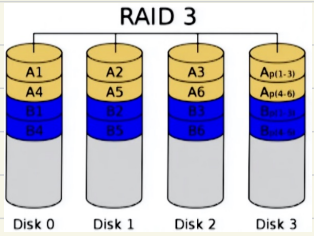
→ RAID 0 : Striping
- ↳ no redundant data
- ↳ least costly
- ↳ reliability is poor
- ↳ best write performance



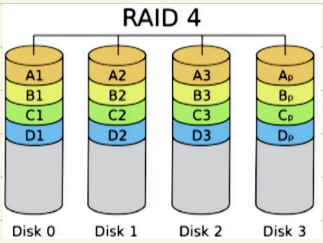
→ RAID 1 : Mirroring
- ↳ two identical copies
- ↳ most expensive
- ↳ excellent fault tolerance
- ↳ space utilisation 50 percent



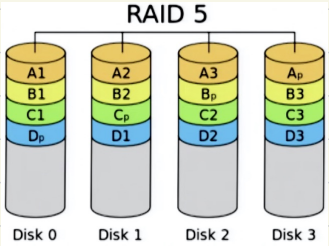
→ RAID 2 : Parity
- ↳ striping unit is single bit
- ↳ Hamming code for parity
  - ↳ For a 4-bit data, 3 bits are added



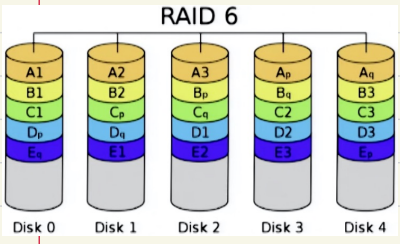
→ RAID 3 : Byte Striping + Parity
- ↳ single check disk with parity information
- ↳ cannot service multiple requests simultaneously



→ RAID 4 : Block Striping + Parity
- ↳ striping unit of a disk block
- ↳ facilitates recovery of at most 1 disk failure
- ↳ write performance low due to the need to write all parity data to a single disk



→ RAID 5 : distributes parity blocks uniformly over all disks
- ↳ several write requests simultaneously
- ↳ read will have high parallelism
- ↳ recovery of 1 disk failure



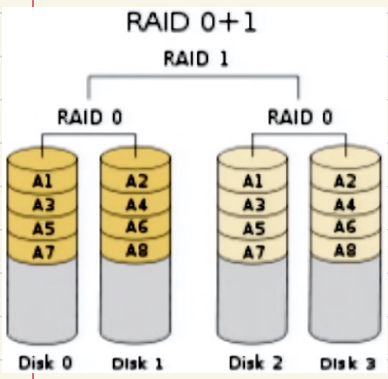
→ RAID 6 : two parity blocks distributed across all member disks
- ↳ write performance poorer than RAID 5
- ↳ recovery of 2 disk failures

$$\begin{array}{r} 1000 \\ 0101 \\ \hline 1101 \\ 1101 \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0100 \\ 0101 \\ \hline 0001 \\ 0101 \\ \hline 0100 \end{array}$$

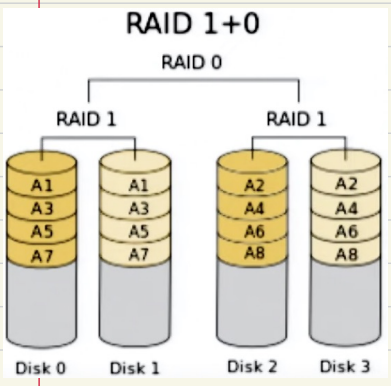
$$01000101 \qquad 01011000 \qquad 01010000$$

69 88 80

→ Hybrid RAID : Nested RAID levels
  ↳ combine two or more
  ↳ example: RAID 50 layers the data stripping of RAID 0 on top of the distributed parity of RAID 5.

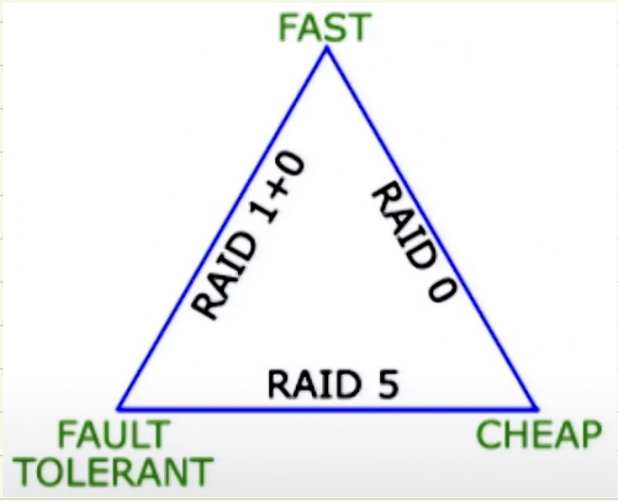


→ RAID 01 : mirror of stripes
  ↳ both replication and sharing of data



→ RAID 10 : Stripe of mirroring
  ↳ mirroring on striped data
  ↳ better throughput and latency than all RAID levels except RAID 0.

→ Choice of RAID levels



→ What does RAID not do?
  → doesn't equate 100% uptime
  → " replace backups
  → " protect against data corruption
  → " allow to dynamically increase array size
  → isn't the best option for virtualisation

Comparison of RAID: Theoretical

PPD

Level	Description	Min.[b] # of drives	Space Efficiency	Fault Tolerance (*Drives*)	Performance (*as factor of single disk*)	
					Read	Write
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: $n-2^{[e]}$

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*

WEEK 12

→ Overview of Query Processing

→ Steps: Parsing/Translator → Optimization → Evaluation

→ Optimizer

↳ ex. SELECT salary FROM instructor WHERE salary < 75000

→ $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ } equivalent plans
$\Pi_{salary}(\sigma_{salary<75000}(instructor))$

↳ amongst all equivalent plans, choose the one with the lowest cost.

→ Measures of Query Cost

→ measured in terms of total elapsed time

↳ disk access is the primary focus

→ number of block transfers from disk and the number of seeks

→ t_T: time to transfer one block

t_S: time for one seek

→ cost of b block transfers plus S seeks = $b \cdot t_T + S \cdot t_S$

→ Selection Operation

A#	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r \times t_T$	One initial seek plus b_r block transfers
A1	Linear Search, Eq. on Key	Average case $t_S + (b_r/2) \times t_T$	Since at most one record satisfies condition, scan can be terminated as soon as the required record is found. b_r blocks transfers in worst case
A2	Prm. Index, Eq. on Key	$(h_i+1) \times (t_T + t_S)$	Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer
A3	Prm. Index, Eq. on Nonkey	$h_i \times (t_T + t_S) + b \times t_T$	One seek for each level of the tree, one seek for the first block. Here all of b are read. These blocks are leaf blocks assumed to be stored sequentially (for a primary index) and don't require additional seeks
A4	Snd. Index, Eq. on Key	$(h_i+1) \times (t_T + t_S)$	This case is similar to primary index
A4	Snd. Index, Eq. on Nonkey	$(h_i+n) \times (t_T + t_S)$	Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large
A5	Prm. Index, Comparison	$h_i \times (t_T + t_S) + b \times t_T$	Identical to the case of A3, equality on nonkey
A6	Snd. Index, Comparison	$(h_i+n) \times (t_T + t_S)$	Identical to the case of A4, equality on nonkey

t_T is time to transfer one block. t_S is time for one seek
b_r denotes the number of blocks in the file
b denotes the number of blocks containing records with the specified search key
h_i denotes the height of the index. n is the number of records fetched

→ Conjunction

↳ A7 conjuctive selection using one index
A8 " " " composite index
A9 " " by intersection of identifiers

Disjunction

↳ A10 disjunctive selection by union of identifiers

↳ applicable if all conditions have available indices otherwise use linear scan

→ Sorting

→ Quicksort : if relation fits in memory

External sort-merge : if relation is too big to fit in memory

→ Join

→ Nested-loop join

↳ $r \bowtie_\theta s$ → for each tuple t_r in r do begin

outer relation → r

inner relation → s

for each tuple t_s in s do begin

test pair (t_r, t_s) to see if they satisfy the join condition θ; if they do, add $t_r \cdot t_s$ to the result

end

end

WEEK 5 GA

① $(P)^+ = PQCAB$ $(B)^+ = BPQCA$ $(C)^+ = C$
$(Q)^+ = QBPCA$ $(A)^+ = AQ$ $(AC)^+ = ACBPQ$

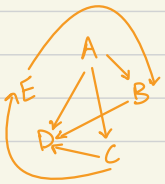
③ R (studinfo (A), courseId (B), name (C), state (D))
superkeys → A, AB, AC, AD
$\# = \sum_{i=0}^{3} {}^3C_i = 1 + 3 + 3 + 1 = 8$

④ $(PR)^+ = PRQST$

⑤ any superkey must have 'A'. prime attributes: A, Y, C, B, X, Z
$(A)^+ = A$ $(AY)^+ = AYZBCX$ $(AB)^+ = ABCXYZ$ $(AZ)^+ = AZBCXY$
$(AC)^+ = ACXYZB$ $(XY)^+ = XYZBC$ $(AX)^+ = AXYZBC$

⑧ if C is extraneous, $F' = A \to C, AB \to D$
$(AB)^+$ w.r.t. F' = ABDC
$(AB)^+$ w.r.t. F = ABCD
if D is extraneous, $F' = A \to C, AB \to C$
$(AB)^+$ w.r.t. F' = ABC
$(AB)^+$ w.r.t. F = ABCD

⑨ $(A)^+ = ABCDE$
since A is not on the R.H.S. of any FD, A must be in every superkey

⑩ 

Since C is only being determined by A, drop A → C.

A → B

2NF : A ⊂ CK AND B non-prime

3NF : A → SK OR B → prime

BCNF : A → SK

WEEK 6 GA

① CK = AB ⇒ BCNF
CK = AB → 1NF
CK = AB ⇒ 2NF
CK = AB, DE ⇒ 3NF

② CK = AB
2NF ✓ 3NF ✗

③ CK = T, PQ
2NF ✓ 3NF ✗

④ IPL(A, B, C, D) FD = { A → BCD, BC → AD, D → B }
CK = A, BC, CD
2NF ✓
3NF ✓
BCNF ✗ because of D → B

(8) R → S ; Q ← P ; R → Q

$D_1 = (P, Q)$, $FD = \{P \rightarrow Q\}$ BCNF ✓

$D_2 = (R, S)$, $FD = \{R \rightarrow S\}$ BCNF ✓

(5) X(A, B, C, D, E, F) CK = A, BC 2NF ✓ BCNF ✓

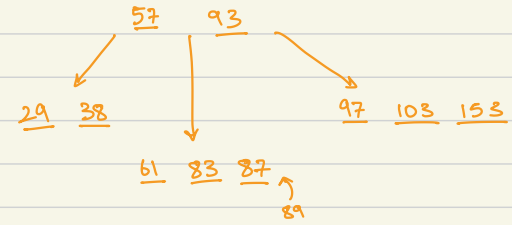
Y(G, H) CK = G 2NF ✓ BCNF ✓

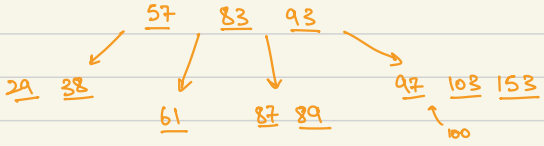
Z(G, A, I) CK = GA 2NF ✓ BCNF ✓

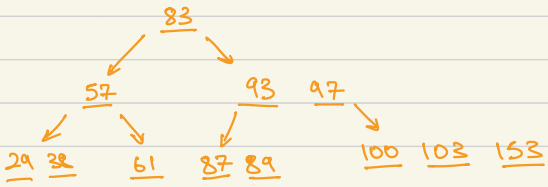
$FD = \{A \rightarrow BCDEF,\ BC \rightarrow ADEF,\ G \rightarrow H,\ GA \rightarrow I\}$

WEEK 9 GA

(9)

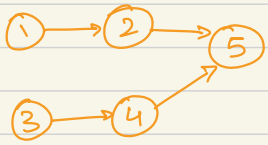






WEEK 10 GA

1	2	3	4	5
				r(Z)
w(Y)				
	r(Y)			
		w(Y)		
			r(Y)	
	w(P)			
				r(P)
			w(X)	
r(Q)				
				r(X)
				w(Y)



serial schedule: 1 2 3 4 5