Data Engineering

1) SQL

-> Case Example. (can count multiple values for diff colours using single query.)

```
select sum(case col1
        when 2.99 then 1
        else 0
        END) as regular,
    sum (case col2
        when 4.99 then 1
        else 0
        END) as rental rate;
```

-> COALESCE(function).

-> Accepts unlimited no. of arguments and returns 1st argument i.e not null if all null returns null.

Eg:- SELECT coalesce(NULL, 1, 2, 3) => 1

★ -> We use COALESCE to substitute null values while doing operations.

Eg:- SELECT items, (price - COALESCE(discount, 0)) As final-price from table1.

Replace discount with value null to 0

~~Cat~~ Cast (operator) :- It lets you convert from one data type to other.

Syntax :- SELECT CAST('5' AS INTEGER)

postgres SQL => SELECT '5'::INTEGER.

-> NULLIF (function) :- Returns null if 2 args passed are '=' else returns 1st arg passed.

-> Adv from table is no phy storage and data queried is up to date

-> View :- A data base obj that is of stored query.

Eg :- CREATE VIEW view_name AS
SELECT c1, c2 from t1 inner join address
on c1.a_id = c2.a_id.

To update view :-
CREAT OR REPLACE view_name AS
query......

Drop view if exists :-
DROP VIEW IF EXISTS view_name.

Rename VIEW :-
ALTER VIEW v1 RENAME ~~[illegible]~~ to v2

-> Importing and exporting data functionality of pg Admin that allows us to import data from .csv file to already existing table.

→ psycopg2 :- Python library ~~to~~ to interact with DB in postgresSQL.

Eg :- import psycopg2 as pg2

```
conn = pg2.connect(database='name', user='pipe'
                   , password='pwd')
cur = conn.cursor()
cur.execute('SELECT * FROM t1')
cur.fetchone() => Fetches query results
cur.fetchmany(10) =>
cur.fetchall()
query = """CREATE TABLE - - - - - - - """
cur.execute(query).
★ cur.commit() => To commit changes to DB.
★ conn.close()
```

Window functions :- They perform aggregate ops on grp of rows but they produce result for each row.

Eg :- SELECT c1, c2, c3, avg(c3) over(partition by c2) as ~~dept~~ c3_avg from t1.

(group by c2)

Eg 2 :- SELECT c1, AVG(c2) over() from emp;

Eg 3 :- select emp_no, dept, sal, count(*) over (PARTITION BY dept) as dept_count. from emp;

*Eg 4 :- SELECT emp_no, dept, sal,
SUM(salary) over (PARTITION BY department
ORDER BY SALARY) as rolling-dept-salary
MIN (salary) over " DESC "

*Eg 5 :- SELECT emp_no, dept, salary,
RANK() over(order by ~~dept~~ salary DESC).

RANK() => Gives rank based on colum value.
*↳ It might miss a rank if their are 3 people with rank 1 it will miss rank 2,3 the next rank will be 4.

DENSE_RANK() → Works like RANK() but ranks will not miss a number if even ~~before~~ preceeding ranks are same

Eg 6 :- SELECT emp_no, dept, salary
NTILE(4) over (order by salary DESC).

Eg 7 :- (FIRST_VALUE), (LAST_VALUE), (NTH_VALUE).
Eg 8 :- (LAG, LEAD), LAG(col1, 2)

★ → We can create table ~~usi~~ from query.

Materialized View :-

→ To avoid having redundant storage and up to date data we use views but for complex queries to execute an view takes time so we use materialized views.

★ → Data is stored physically.

→ Eg 1 :- CREATE MATERIALIZED VIEW v_name
As query.

Eg 2 :- REFRESH MATERIALIZED VIEW
(To update view with new data from the view query)

→ we can update data manually (or) use triggers.

→ Eg 3 :- ALTER MATERIALIZED VIEW
DROP MATERIALIZED VIEW

(named subquery)
Common table expr (CTE) :-

→ To use query result in other query.

★ Syntax :- With cte_name AS (
SELECT ......)
SELECT ...... cte_name ...;

Multi CTE :-

Syntax :- with name1 as (   ),
name2 as (   )
Select - ----.

-> We can not use aliases ~~as~~ for colname in same query level.

Indexing :- -> Single level (primary, clustered, Secondary).
-> Multi leve (BTree, B+tree).

-> It reduces I/O cost. (It reduces time take to data transfer b/w RAM and hard disk.)

Indexing types.
-> We use $\boxed{\text{Sparse}}$ when data is sorted
-> Here ~~only~~ the whole block pointer is stored.
-> We use $\boxed{\text{Dense}}$ when data/records is unsorted
-> Here the ~~pointer is~~ each row pointer is stored.

-> Primary Indexing :-
-> Ordered file where records (key + pointer) of fixed length.

-> Clustering Indexing :-
-> Records are physically ordered on non-key field which doesn't have distinct value for each record
~~For each block of disti~~
-> Each distinct value of non-key attribute an entry is created.
-> ~~Each~~ for Duplicate non-key attribute ~~points to~~ single pointer is used for reference.

Secondary Indexing:-

-> Creating index on 2nd attribute when we have primary indexing.

-> 2nd attr is not ordered in DB. But can order it in index table.

1° index syntax:-

-> CREATE INDEX index_name
   on table_name (col_name);
   (col1, col2); multi col index.

-> DROP INDEX index_name;

-> Indexing should be reconsidered when:-
   -> Small tables
   -> Have frequent large batch update, insert op's.
   -> On columns with high no. of null values.
   -> Columns that are frequently manipulated.

-> Execute sp_helpindex table_name; => on particular table. * To see all the indexes

-> In SQL server -> Clustered index. / -> Non-clustered index.

-> Clustered index:-
   -> Defines the order in which data is physically stored in a table. As ~~order~~ sorting can be done in only way we have one clustered index per table. In SQL server 1° key constraint auto creates

clustered index.

-> When new data is added it maintains order based on cluster index value.

-> Custom clustered index!-

```
CREATE CLUSTERED INDEX index_name
on table_name (col1, col2);
```

-> Non clustered index!-

-> Separate storage is needed.

-> Can have multiple non-clustered indexes per table.

->

Types of DB!- 1. key-value, 2. wide column, (used when freq writes and not much read and update).
(Redis, memcache) (Cassandra, Hbase)
Schemaless, CQL

3. Document DB (MongoDB, Firestore, DynamoDB, couchDB)

-> Unstructured, No schema.

-> Collection (contain 2 or more documents).

-> Document (contain key-value pairs JSON).

* -> Fields within collection can be indexed and collections can be organized into a logical hierarchical

-> Reads are much faster while write and update is complex.

4. RDB !- (MySQL, PostgreSQL, SQL server...).

-> ACID compliant.

-> Cockroach DB scales horizontally without reconfig.

5. Graph DB :- (Neo4J, Dgraph).

→ Nodes contain data.

→ Edges defines relationship

→ Good choice alt for RDB if we have lot of joins and perf issue.

6. Search Engine :- (Give list of most appropriate results).

→ Most are on top of Apache Lucene

Eg:- Solr, elastic search.

7. MultiModel (FAUNA) :-

→ Use GraphQL to describe how we want data

MongoDB :- (Document DB) Database ⇄ Collections ⇄ documents.

→ Documents can be stored in formats like JSON, BSON, XML.

→ BSON :- Binary JSON to overcome JSON drawback

↳ It encodes type and length information Makes it possible to traverse more quick compared to Json.

↳ It adds some non-JSON data types (date, binary data)

↳ MongoDB driver converts Json ⇄ BSON.

→ Some Document DB provide schema validation.

→ Collections :- Set of documents that can have different schema. Eg:- Users collection can have 2 documents with 1 user detail in each doc.

→ CRUD can be done here.
↳ Using unique id / field values. Indexes can be used for good perf.

→ Document DB's are distributed. (Horizontal scalling).

→ Single transaction works fine in MongoDB.

→ ~~for mult~~ MongoDB support multi transactions but the perf might be affected.

→ Cluster is place where MongoDB DB's are stored.

★ → Database design and data modeling are major factors in MongodB performance.

★ → ~~here~~ _id will be created automatically for each new doc created in collection. But we can assign it manually using "_id" as key.

★ → $ with word in mongoDB indicate its reserved word.

→ db.flightData.updateMany ({ }, {$set: {mark: "delete"}})
collection — filter set to empty to update all docs — if key exist it will update value else add it.

→ db.flightData.insertMany ([{},{}])
pass ~~collection~~ documents here.

→ db.flightData.find({distance: {$gt: 10000}}).pretty().
(to get data distance > 10000)

→ db.flightData.update({_id: ObjectId('xxx')}, {dela: false})
This will not just set the dela field to false it will replace whole obj with only _id and dela keys.

→ db.passengers.find() ⇒ Returns cursor to go through elements.

→ db.passengers.find().toArray() ⇒ Returns all elements (documents)

→ db.passenger.find().forEach() ⇒ Returns 1 element for a loop reducing load...

→ <u>Projection</u> :- Instead of fetching whole document (or) object and filtering it in app which increase load / latency we can filter (or) get the data only we want ~~[illegible]~~ by filtering within DB we use <u>projection</u>.

<u>Eg</u>:- db.passengers.find({}, {name: 1}) ⇒ Returns all data with id, name

db.passengers.find({}, {name: 1, _id: 0}) ⇒ Returns all data with only name

→ <u>Embedded Document</u> :- Document inside other document.

→ Max document size in MongoDB is 16MB.

→ db.hobbies.insertMany([{n': 1}, ...], {ordered: false});

→ By default ordered is true so if any doc insertion fails only preceeding docs would get inserted to avoid it we can set to false

→ We use journal ~~in case~~ to maintain [to-do] tasks so that in case the memory ~~will~~ get wiped out (power cut) it still knows what to do using journal. {j: undefined (no response if it saved in journal), j: True (Response)

→ <u>Comment in shell</u> :-

```
mongoimport tv-shows.json -d movieData -c movies --jsonArray --drop
```

-d → db name

-c → collection name

--jsonArray → need to be specified if list of docs in file.

--drop → if exist: with del + create, without: append