Data 1050

Xiaoke Song

1 Data and Operations

1.1 4 Types of Data

- **Nominal Data**: categorical, lack order or ranking; eg: gender, nationality, hair color; **frequency table, bar charts**.
- **Ordinal Data**: with order or ranking, but differences not quantifiable; eg: customer ratings, educational level, survey responses; **bar charts, pie charts**, central tendency.
- **Discrete Data**: numbers; eg: number of employees; **histograms, bar charts**.
- **Continuous Data**: range or interval such as time, temperature, or distance; eg: height or weight of an object, speed of a car; **histograms, scatterplots, line charts** to identify trends or relationship.

1.2 Relation Model

- **Definition**: **Attributes** are the unique columns of data, **Tuples** are rows of data. **Relation** contains Schema (metadata $\rightarrow$ what each columns and rows represents) and Instance (data itself $\rightarrow$ content).
- In a **Relation**, the **Tuples** should fit the table Schema; and the **Attributes** should be unique.
- **Definition**: the **primary key** is the unique identifier (only one per table); the **foreign key** helps connect with other tables.

stu_id	name	class_id
123	Max	3
234	Kendra	2
345	Adam	1
456	Lily	3

Table 1: Students Table

id	name	location
1	DS	Rm 101
2	ML	Rm 301
3	Stat	Rm 201

Table 2: Classes Table

- The **primary key** in students table is `stu_id`, in classes table is `id`. The **foreign key** in students table is `class_id`, in classes table is `id`. The foreign key in students table is the primary key in classes table.
- **Definition**: A **superkey** is any <u>combination</u> of attributes that can uniquely identify a record, but it might include extra information that isn't necessary. The **primary key** is a <u>minial</u> superkey.
- **Definition**: A **candidate key** (can contain NULL values) is a set of attributes uniquely identifying rows in a table. **Uniqueness:** No two rows have same value; **Minimality:** No subset can uniquely identify rows. Candidate keys can be multiple, they are potential choices of primary keys.
- The decision of primary key (PK) depends on the rules of different models. eg: Enrollment: {Student_id, Course_id, Section, Semester}:
  - {Student_id} is PK: each student can only take one course, one section in one semester.
  - {Student_id, Course_id} is PK: each student can take <u>multiple</u> courses, but can only take one specific course <u>once</u>.
  - {Student_id, Course_id, Section} is PK: each student can take <u>multiple</u> sections of a course in the <u>same</u> semester.
  - {Student_id, Course_id, Semester} is PK: each student can take <u>same</u> course in <u>different</u> semester, but <u>same</u> section.
  - {Student_id, Section, Semester} is PK: each student can take <u>multiple</u> courses in one semester, but each course should have <u>unique</u> section number.

2 SQL

2.1 Cheat Sheet

```
SELECT [DISTINCT] column_expression_list [AS] alias
FROM table_name
 [[INNER | LEFT] JOIN table_name ON qualification_list]
 [WHERE search_condition]
 [GROUP BY column_name]
 [HAVING search_condition]
 [ORDER BY column_list]
 [LIMIT number_of_rows]
 [OFFSET number_of_rows];
```

- **Description:**
  - `SELECT`: List is comma-separated. Column expressions may include aggregation functions. `DISTINCT` selects only unique rows.
  - `WHERE a IN cons_list`: Select rows for which the value in column a is among the values in a `cons_list`.
  - Use `WHERE` to filter data <u>before</u> aggregation (`JOIN`, `GROUP`). Use `HAVING` to filter data <u>after</u> aggregation.
  - `ORDER BY a [ASC], b DESC`: Order by column a (ascending by default), then b (descending).
  - `OFFSET number`: Skip the first number rows in the return result.

2.2 SQL in Data Definition Language (DDL)

- **Create a table:**

```
CREATE TABLE instructor (
                    ID CHAR(5),
                    name VARCHAR(20) NOT NULL,
                    dept_name VARCHAR(20),
                    salary NUMERIC(8,2) CHECK (salary > 0.0),
                    UNIQUE (name, dept_name),
                    PRIMARY KEY (ID),
                    FOREIGN KEY (dept_name) REFERENCES department(dept_name),
                    ON DELETE CASCADE,
                    ON UPDATE CASCADE,
                    ON INSERT CASCADE
                    );
```

  - The `PRIMARY KEY` constraint is important because (1) **Uniqueness**: it prevents duplication by disallowing the insertion of rows where the primary key value already exits. (2) **Non-NULL**: a primary key attribute cannot contain NULL.
  - Using `FOREIGN KEY` constraint in example above, we define the first `dept_name` to be the foreign key in dataset `instructor`, then we reference the second `dept_name` to be the primary key in dataset `department`.
  - `NOT NULL` constraint specifies that `name` cannot take null values.
  - The `CHECK` constraint ensures that only values greater than 0.0 are allowed in the `salary` column; if an attempt is made to update or insert a row with `salary` that is less or equal to 0.0, the dataset will reject the operation and return an error.
  - Using `UNIQUE`(name, dept_name) ensures no two instructors with the same name exist in the same department, which specifies <u>{name, dept_name} form a candidate key</u>.
  - `CASCADE` on foreign keys define how changes in the referenced table (<u>department</u> in my case) propagate to the table with the foreign key (<u>instructor</u> here).

- **Alters to attributes:**

```
ALTER TABLE instructor
ADD CONSTRAINT PK_instructor  #need to name the constr
PRIMARY KEY (ID);
```

- **Views in SQL:**
  We use `VIEW` to simplify complex SQL queries and provide restriction to users from accessing sensitive data.

```
CREATE VIEW statistics_students AS
SELECT student_id, name
FROM student_table
WHERE class_id = 3;    # We use ';' to separate multiple SQL statements

SELECT name
FROM statistics_students
WHERE student_id = 123;

# which is equivalent to:
SELECT name
FROM student_table
WHERE student_id = 123 AND class_id = 3;
```

  Equivalent expression using subqueries:

```
SELECT name
FROM (SELECT student_id, name
      FROM student_table
      WHERE class_id = 3) AS statistics_students
WHERE student_id = 123;
```

stu_id	name	class_id
123	Max	3
234	Kendra	2
345	Adam	1
456	Lily	3

Table 3: Students Table

$\longrightarrow$

name
Max

Table 4: Output

- **More examples of Subqueries:**
  Find courses offered in Fall 2021 and Spring 2022 (subquery in `WHERE` clause):

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND years = 2021 AND
     course_id IN
              (SELECT course_id
               FROM section
               WHERE semester = 'Spring' AND years = 2022) AS spring_2022;
```

  - We can use `NOT IN` if asked to find course offered in Fall 2021 but not in Spring 2022.
  - Subqueries are required to have names, which are added after parentheses the same way you would add an alias to a normal table.

  Find the average instructors' salaries of those departments where the average salary is greater than $80,000 (subquery in `FROM` clause):

```
SELECT dept_name, avg_salary
FROM (SELECT dept_name, AVG(salary) AS avg_salary
      FROM instructor
      GROUP BY dept_name) AS group_dept
WHERE avg_salary > 80000;
```

  Find the number of instructors in each department (subquery in `SELECT` clause):

```
SELECT dept_name, (SELECT COUNT(*)
                   FROM instructor
                   WHERE department.dept_name = instructor.dept_name)
                   AS num_instructors
FROM department;
```

- **Temporary Tables:**
  Create tables not stored in database, but exists only while in database session where it created:

```
CREATE TEMPORARY TABLE temporary_table (stu_id INT, name VARCHAR);

INSERT INTO temporary_table VALUES (123, 'Adam'), (234, 'Lily');
```

  – Temporary tables can be modified using `UPDATE` and `INSERT`.

- **Drop Table:**
  Drop table instructors:

```
DROP TABLE instructors;
```

- **Procedures and Functions:**
  Write a procedure to retrieve the total credits for a specific student, based on their studnet ID, and stores the result in an output parameter:

```
DELIMITER //
CREATE PROCEDURE studentcount(
        IN sid VARCHAR(128),      # 'input' parameter
        OUT total DECIMAL(3,2))   # 'output' parameter
BEGIN
    SELECT tot_cred INTO total
    FROM student
    WHERE student.id = sid;
END //

DELIMITER ;
CALL studentcount('1122', @total);  # out parameter requires @ to store
```

  – `DELIMITER //` changes the statement delimiter from `;` to `//` so that the procedure can contain multiple SQL statements.
  – The `studentcount` is stored in the database and can be called any time by passing a specific `id` as a parameter. Executing "`SELECT` @total;" will display the value stored in "@total", which represents the total credits of the student with ID `1122`.

  We can write a function that does the same thing:

```
DELIMITER //
CREATE FUNCTION studentcount(sid VARCHAR(128))
RETURNS DECIMAL(3,2)
BEGIN
    DECLARE total DECIMAL(3,2);   # creating variable
    SELECT tot_cred INTO total
    FROM student
    WHERE student.id = sid;
    RETURN total
END //

DELIMITER ;
SELECT studentcount('1122');
```

  – The function `studentcount` is stored in the database and can be invoked any time with a specific value for `id`.

2.3 SQL in Data Manipulation Language (DML)

- **Insert data:**
  Example of inserting data followed after `VALUES` as a new row into the `course` table:

```
INSERT INTO course VALUES
('Data1050', 'Data-Engineering', 'DSI', NULL);
```

  or equivalent:

```
INSERT INTO course (course_id, title, dept_name, credits)
VALUES ('Data1050', 'Data-Engineering', 'DSI', NULL);
```

- **Updates to data:**
  In `instructor` table, those `salary` less than 5000 be updated to their `salary` increased by 50%:

  ```
          UPDATE instructor
          SET salary = salary*1.5
          WHERE salary < 50000;
  ```

  - We use `ALTER` to update the attributes. We use `UPDATE` to change the data.

- **Deletes data:**
  Remove rows in `instructor` where the `salary` is greater than 75000:

  ```
              DELETE FROM instructor
              WHERE salary > 75000;
  ```

- **Set Operations:**
  If two tables have the same number of attributes and the type of attributes in corresponding positions are same (don't have to have same name), then we call them "set operation compatible" and we can do set operations on them.
  Find courses that were offered in Fall 2020 **or** in Spring 2021:

  ```
     (SELECT course_id FROM section WHERE sem = 'Fall' AND years = 2020)
    UNION (SELECT course_id FROM section WHERE sem = 'Spring' AND years = 2021);
  ```

  Find courses that were offered in Fall 2020 **and** in Spring 2021:

  ```
     (SELECT course_id FROM section WHERE sem = 'Fall' AND years = 2020)
    INTERSECT (SELECT course_id FROM section WHERE sem = 'Spring' AND years = 2021);
  ```

  Find course that were offered in Fall 2020 **but not** in Spring 2021:

  ```
     (SELECT course_id FROM section WHERE sem = 'Fall' AND years = 2020)
    EXCEPT (SELECT course_id FROM section WHERE sem = 'Spring' AND years = 2021);
  ```

  - Set operations automatically eliminate duplicates. To retain duplicates, we could use `UNION /INTERSECT/EXCEPT ALL`.

- **Common Table Expressions (CTEs):**
  CTEs function like `VIEW` but in one single query; unlike `VIEW`, CTEs will not be stored:

  ```
     WITH statistics_student AS (
          SELECT student_id , name
          FROM student_table
          WHERE class_id = 3)    # without ';'

    SELECT name
    FROM statistics_student
    WHERE student_id = 123;
  ```

- **Triggers:**
  Triggers are used when handling actions like enforcing rules or updating data when certain events occur, eliminating the need for manual updates and ensuring consistency across the database.

  Automatically change any attempted update of a student's grade in the `takes` table to `NULL` if the grade is not one of the allowed values ('A', 'B', 'C', 'D', 'F', 'INC'):

  ```
    CREATE TRIGGER setnull
         BEFORE UPDATE ON takes
         REFERENCING NEW ROW AS nrow    # define alias after updated row of takes
         FOR EACH ROW
         WHEN nrow.grade NOT IN ('A', 'B', 'C', 'D', 'F', 'INC')
    BEGIN
         SET nrow.grade = NULL
    END;
  ```

  - `BEFORE` and `AFTER` specify whether the trigger action happen <u>before</u> or <u>after</u> the database operation (eg: `INSERT`, `UPDATE`, `DELETE`).

- **Window Functions:**
  Window Functions are used within `SELECT` statements to perform calculations across a set of rows related to the current row without collapsing the result into a single value, unlike aggregate functions.

  Insert the average salary for each position in each row for that position:

```
SELECT job_title, salary,
    AVG(salary) OVER (
    PARTITION BY job_title   # divide to partitions (groups) based job_title
    ) AS avg_salary
FROM salary_info;
```

job_title	avg_salary
ANALYST	3000
SALES	2100
ENGINEER	3500

Table 5: GROUP BY

job_title	salary	avg_salary
ANALYST	3100	3000
ANALYST	2900	3000
SALES	1600	2100
SALES	2200	2100
SALES	2500	2100
ENGINEER	3500	3500

Table 6: Window Function

Rank the unit prices of items ordered by each customer:

```
SELECT O.customerID, O.orderdate, OD.unitprice,
 RANK() OVER (
    PARTITION BY O.customerID
    ORDER BY OD.unitprice DESC) AS UnitRank
 DENSE_RANK() OVER (
    PARTITION BY O.customerID
    ORDER BY OD.unitprice DESC) AS Unit_dense_Rank
FROM orders As O
INNER JOIN orderdetail AS OD
ON O.id = OD.orderID;
```

  - `RANK()` assigns a rank column to each row within a partition. If there are ties, rows with the same value receive the same rank, but the next rank is skipped. We can avoid rank gaps problem using `DENSE_RANK()`.

customerID	orderdate	unitprice	UnitRank	Unit_dense_Rank
C001	2023-01-01	15.00	1	1
C001	2023-01-01	15.00	1	1
C001	2023-01-01	10.00	3	2
C002	2023-01-05	25.00	1	1
C002	2023-01-05	20.00	2	2

Create a new column that provides the previous order date's `quantity` for each `productID`:

```
SELECT O.productID, O.orderdate, OD.quantity,
 LAG(quantity) OVER (
    PARTITION BY productID
    ORDER BY orderdate) AS Lag
 LEAD(quantity) OVER (
    PARTITION BY productID
    ORDER BY orderdate) AS Lead
FROM orders AS O, orderdetail AS OD
WHERE O.id = OD.orderID;
```

  - `LAG`(quantity) retrieves the value of `quantity` column from the previous row in the partition, which causing the first row `NaN`. We can substitute it to `LEAD`(), which retrieves the value from a subsequent row in the result set, leading the last row `NaN`.

productID	orderdate	quantity	Lag	Lead
P001	2023-01-01	10	NaN	15
P001	2023-02-01	15	10	25
P001	2023-03-01	25	15	NaN
P002	2023-01-15	20	NaN	NaN

Compute sum of all preceding ages per partition:

```
SELECT id, location, age, SUM(age) OVER(
     PARTITION BY location
     ORDER BY age
     RANGE BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING)
     AS a_sum
FROM stops
ORDER BY location, age
```

- `UNBOUNDED PRECEDING` includes all preceding rows in the partition up to the current row.
- `1 PRECEDING` excludes the current row and only includes rows before it. This means the sum is calculated using all rows from the beginning of the partition up to the row immediately preceding the current row.
- We can operate different range with `UNBOUNDED PRECEDING`, `UNBOUNDED FOLLOWING`, `CURRENT ROW`, `number PRECEDING`, `number FOLLOWING`.

id	location	age	a_sum
1	A	10	NULL
2	A	15	10
3	A	20	25
4	B	12	NULL
5	B	18	12

2.4 SQL in Data Query Language (DQL)

- **SQL query:**
  A query to retrieve the names of all Comp. Sci. instructors whose salary is greater than 180,000:

```
SELECT name              #the columns to be returned
FROM instructor          #which tables
WHERE dept_name = 'Comp.Sci.' AND salary > 180000;
```

  - The filter condition we used (after WHERE operation, including `dept_name` and `salary`) don't necessarily have to be in the SELECT statement (including `name`).
  - We will get an empty table instead of error if no data meet the filter condition.

Generates a new table using cartesian product (cross-product), containing every possible combination of rows from two tables:

```
SELECT *
FROM instructor, teaches
WHERE instructor.ID = teaches.ID;
```

Divide salary by 12 and rename the column as monthly_salary:

```
SELECT ID, name, salary/12 AS monthly_salary
FROM instructor;
```

Converting their runtime from minutes to hours:

```
SELECT premier, CAST(runtime_minutes AS DOUBLE PRECISION)/60 AS runtime_hours
FROM titles
WHERE premier >= 2020
AND premier <= 2023;

# equivalent format
runtime_minutes * 1.0 /60 AS runtime_hours
```

  - `DOUBLE PRECISION` specifies that the number should be stored as a double-precision floating-point.

Ordering table by name attribute with ascending/descending order:

```
SELECT name
FROM instructor
ORDER BY name [ASC | DESC]; # can sort multiple attributes
```

Find the name of instructors whose salary is between 90000 and 100000:

```
SELECT name
FROM instructor
WHERE salary BETWEEN 90000 AND 100000;
```

  - whether the start and end are included using `BETWEEN` depends on SQL's favor, so it is more efficient to use `salary` >= 90000 `AND salary` <= 100000.

Find the names of all instructors who have a higher salary than some instructor in `Comp. Sci`:

```
SELECT DISTINCT T.name
FROM instructor AS T, instructor AS S
WHERE T.salary > S.salary AND S.dept_name = 'Comp.Sci.';

# Equivalent subqueries form:
SELECT DISTINCT name
FROM instructor
WHERE
 salary > SOME(SELECT salary FROM instructor WHERE dept_name = 'Comp.Sci.')
 AS comp_sci_salary;
```

  - We can join a table to itself, like table `instructor` above.
  - `SOME` check any at least one salary on the right table fulfills the condition. We can use `ALL` instead to find the instructors who have a higher salary than all instructor in `Comp. Sci`.

Find the IDs of tweet that the number of characters used in its `content` is strictly greater than 15:

```
SELECT id
FROM tweets
WHERE LENGTH(content) > 15;
```

Find IDs with higher temperatures compared to yesterday:

```
SELECT DISTINCT id
FROM weather AS a, weather AS b
WHERE a.temp > b.temp
AND DATEDIFF(a.record_date, b.record_date) = 1;
```

- **CASE WHEN Syntax:**
  Compute the age of all people. Define age as: If they are still alive in 2024, then their current age. Otherwise, the age at which they passed:

```
SELECT person_id, name, died, born
    CASE WHEN died IS NULL
            THEN 2024 - born
         ELSE died - born
    END AS age
FROM people;
```

`CASE WHEN` can go in the `ORDER BY`, `WHERE`, `IN`, and `HAVING` clause:

```
# ordering people their age
SELECT person_id, name, died, born
FROM people
ORDER BY (CASE WHEN died IS NULL
                    THEN 2024 - born
                   ELSE died - born
          END)
DESC;
```

- **String Operations:**
  Find the names of all instructors whose name includes the substring: `dar`:

```
        SELECT name
        FROM instructor
        WHERE name LIKE '%dar%';
```

  - The % character matches any substring. If we want to match `100 %`, using `100 \%`.
  - If we change to `_dar_`, then it matches only names where `dar` is preceded and followed by exactly one character (eg: Adara, non-eg: Edarwin).

- **Aggregate Functions:**
  Find the total number of accounts whose balance is greater than 500.

```
    SELECT COUNT(*)     # AVG(), MIN/MAX(), SUM()
    FROM account
    WHERE balance > 500;
```

  - `COUNT(*)` counts all the rows in the table, regardless of whether any columns have `NULL` values. `COUNT(attribute)` counts only the rows where column `attribute` is not `NULL`.

- **Sampling:**
  Three common techniques from sampling rows from a relation:

```
    # sampling randomly
    SELECT * FROM a_table
    ORDER BY RANDOM()
    LIMIT 10;

    # sampling probability
    SELECT * FROM a_table
    TABLESAMPLE BERNOULLI(p)   # p=1 means 1%

    # sampling system page-level
    SELECT * FROM a_table
    TABLESAMPLE SYSTEM(p)
    # include each memory page rather than individual row with probability p%
```

- **Group By:**
  Find the average salary of instructors in each department:

```
  SELECT dept, AVG(salary) AS avg_salary
  FROM instructor
  GROUP BY dept
  HAVING AVG(salary) > 50000;
```

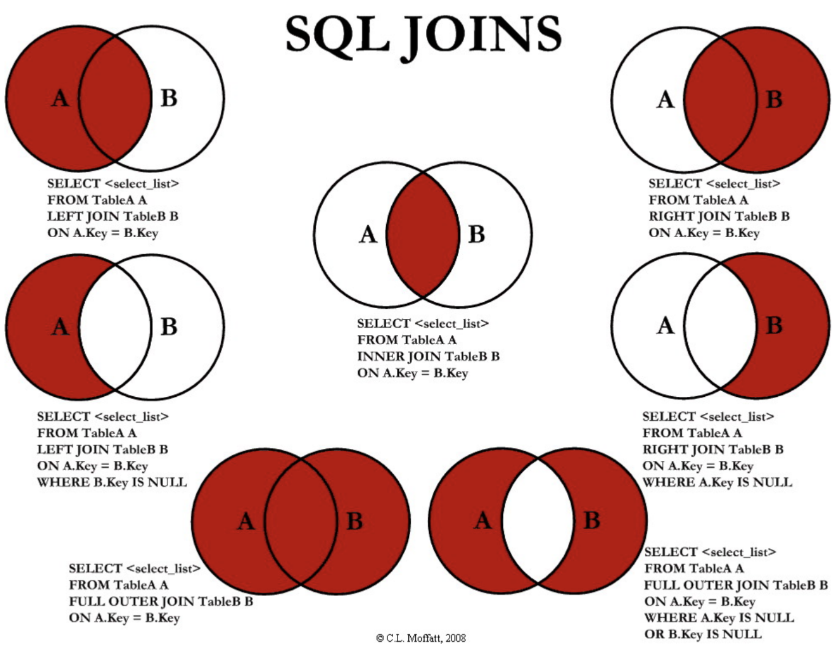
  - Although we use alias in `SELECT` clause, we could not use alias in the rest clause like `GROUP BY`, we have to use its original name.

ID	name	dept	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Brandt	Comp. Sci.	92000
83821	Soni	Comp. Sci.	65000
12121	Wu	Finance	90000
76543	Singh	Finance	90000

$\longrightarrow$

dept	avg_salary
Biology	72000
Comp. Sci.	77333
Finance	85000

- **SQL Joins:**



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

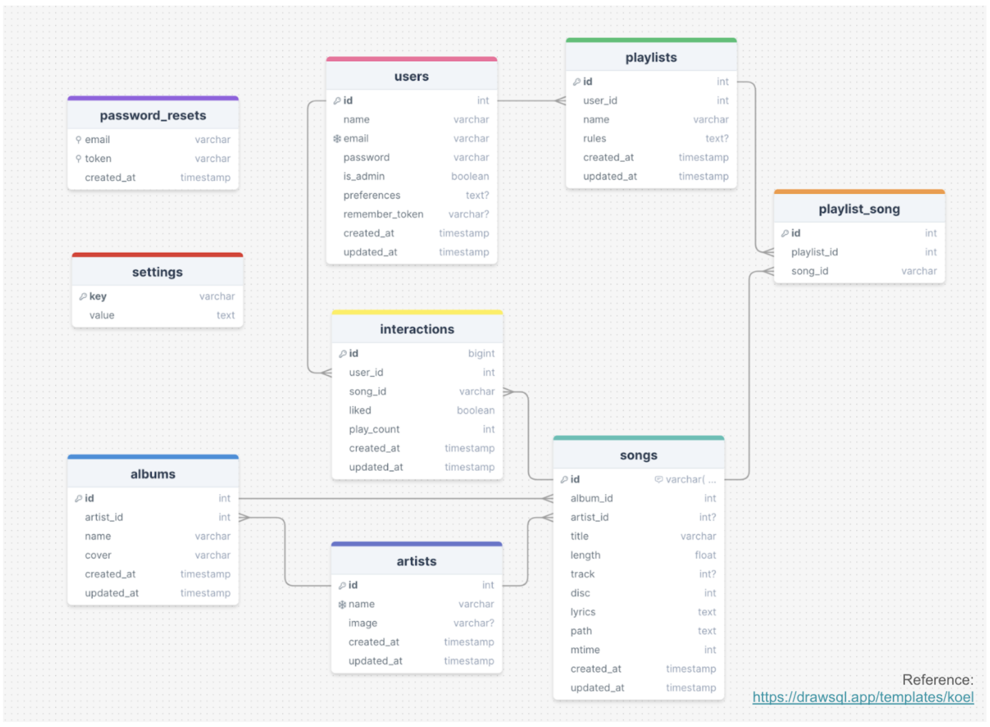
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

2.5 In-class Example

Question: Find playlists that match your interests given the database below.



password_resets	
email	varchar
token	varchar
created_at	timestamp

settings	
key	varchar
value	text

albums	
id	int
artist_id	int
name	varchar
cover	varchar
created_at	timestamp
updated_at	timestamp

users	
id	int
name	varchar
email	varchar
password	varchar
is_admin	boolean
preferences	text?
remember_token	varchar?
created_at	timestamp
updated_at	timestamp

interactions	
id	bigint
user_id	int
song_id	varchar
liked	boolean
play_count	int
created_at	timestamp
updated_at	timestamp

artists	
id	int
name	varchar
image	varchar?
created_at	timestamp
updated_at	timestamp

playlists	
id	int
user_id	int
name	varchar
rules	text?
created_at	timestamp
updated_at	timestamp

playlist_song	
id	int
playlist_id	int
song_id	varchar

songs	
id	varchar(...
album_id	int
artist_id	int?
title	varchar
length	float
track	int?
disc	int
lyrics	text
path	text
mtime	int
created_at	timestamp
updated_at	timestamp

Reference:
https://drawsql.app/templates/koel

Strategy:

1. Create a `VIEW` of songs liked by users in the last week by performing an `INNER JOIN` of `users` to the `interactions` table
2. Use a `CTE` to identify the playlist id's of all the above songs by performing an `INNER JOIN` to the `playlist_song` table
3. Using a `LEFT JOIN`, find all the playlist names for each playlist id (and filling the nulls to `Unknown` using `COALESCE`) to recommend to users

Answer:

```
CREATE VIEW liked_songs AS
SELECT A.id AS user_id, B.song_id
FROM users A
INNER JOIN interactions B
ON A.id = B.user_id
WHERE liked = TRUE                    # filter interactions
AND created_at >= "2024-05-08";

 WITH liked_playlist_ids AS (
         SELECT A.user_id, C.playlist_id
         FROM liked_songs A
         INNER JOIN playlist_song C
         ON A.song_id = C.song_id)

SELECT A.user_id, COALESCE (B.name, 'Unknown') AS playlist_name
FROM liked_playlist_ids A
LEFT JOIN playlist B
ON A.playlist_id = B.id;
```

3 Functional Dependency & Normalization

3.1 Function Dependency

Definition: A functional dependency means that when the value of one attribute is known, one can always determine the value of another attribute. We write A `->` X, where A is called Determinant, and X is called Dependent.

3.2 Decomposition of Schema

The process of breaking up a single relation into two or more sub relations to avoid Redundancy/Insertion/Update/Deletion Anomaly.

- Example FD: {dept, building} `->` P_lot

emp_id	dept	building	P_lot
11	Bio	Watson	A23
22	CS	Mudd1	B45
33	CS	Mudd2	B46

Table 7: Original Table

$\longrightarrow$

emp_id	dept
11	Bio
22	CS
33	CS

dept	building	P_lot
Bio	Watson	A23
CS	Mudd1	B45
CS	Mudd2	B46

Table 8: Wrong Decomposition

If we join two sub tables, we will create incorrect relation as there are four combination with `dept` is equal to `CS` but only two of them stands. That is why we need correct decomposition below:

emp_id	dept	building	P_lot
11	Bio	Watson	A23
22	CS	Mudd1	B45
33	CS	Mudd2	B46

Table 9: Original Table

$\longrightarrow$

emp_id	dept	building
11	Bio	Watson
22	CS	Mudd1
33	CS	Mudd2

dept	building	P_lot
Bio	Watson	A23
CS	Mudd1	B45
CS	Mudd2	B46

Table 10: Correct Decomposition

3.3 Types of Satisfaction of a FD

- **Trivial Satisfaction:** For A `->` X, if X is entirely contained in A (eg: {A,B,C} `->` {A,B}), or the relation R does not contain all the attributes on (A or X) (eg: R = {A,B,C,D,E}, {A,B} `->` {E,F}).
- **Non-trivial satisfaction:** The relation is not obvious or trivial. By the law of transitivity, {A} `->` {B} and {B} `->` {C} implies {A} `->` {C}.

3.4 Normalization

- **First Normal Form (1NF):** A table is in 1NF if:
  1. A single cell must not hold more than one value.
  2. There must be a primary key for identification.
  3. Each column must have only one value for each row in the table.
  4. No duplicated rows or columns.

Player	Item
goldboy10	1 sword, 5 shields
knight123	3 swords, 4 amulets, 1 shield, 7 horses
gotback1	2 swords

Table 11: Original Table

$\longrightarrow$

Player	Item	cnt
goldboy10	sword	1
goldboy10	shield	5
knight123	sword	3
knight123	shield	1
knight123	amulet	4
knight123	horses	7
gotback1	sword	2

Table 12: Following 1NF

- **Second Normal Form (2NF):** A table is in 2NF if:
  1. It is already in 1NF.
  2. It has no partial dependency - that is, all non-key attributes are fully dependent on the entire primary key.

Player	Item	cnt	player_rating
goldboy10	sword	1	beginner
goldboy10	shield	5	beginner
knight123	sword	3	intermediate
knight123	shield	1	intermediate
knight123	amulet	4	intermediate
knight123	horses	7	intermediate
gotback1	sword	2	advanced

Table 13: Counter-Example

In this table, the primary key is {Player, Item}, but the attribute `player_rating` only partial depends on `Player` ({Player} -> {player_rating}), not the entire primary key, violating 2NF.

- **Third Normal Form (3NF):** A table is in 3NF if:
  1. It is already in 2NF.
  2. No transitive partial dependency - that is, no non-prime (non-key) attribute (an attribute that is not part of the candidate key) is transitively dependent on the primary key.

Player	player_rating	skill_level
goldboy10	beginner	1
knight123	intermediate	4
gotback1	advanced	8

Table 14: Counter-Example

Mapping `skill_level` to `player_rating` like this: 1-3 - beginner; 4-6 - intermediate; 7-9 - advanced

In this table, {Player} -> {skill_level} -> {player_rating} (transitive dependency), which violates 3NF. Here, `player_rating` depends on `skill_level`, not directly on `player`, meaning that if you update `knight123`'s `skill_level` from 4 to 7, you will also need to update his `player_rating` from intermediate to advanced. In this case, we can do decomposition of schema.

- **Boyce-Codd Normal Form (BCNF):** A table is in BCNF if:
  1. It is already in 3NF.
  2. For every functional dependency {X} -> {Y}, {X} must be a superkey.

Year	**Rank**	**Title**	**year_mo**
2020	1	Round	2020-04
2020	2	Metal	2020-01
2020	3	Land	2020-10
2021	1	Spider	2021-05
2021	2	Dune	2021-11
2021	3	Coda	2021-06

Table 15: 3NF Table

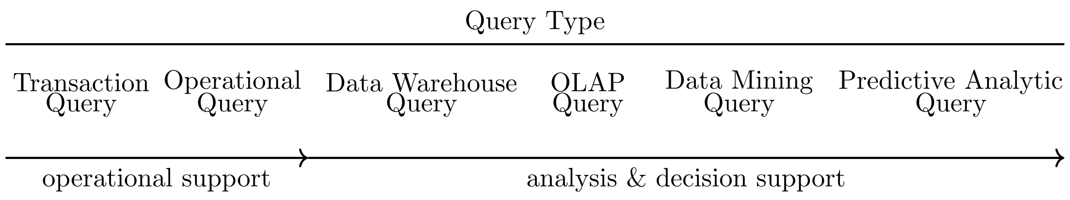
Year	**Rank**	**Title**	**Month**
2020	1	Round	April
2020	2	Metal	January
2020	3	Land	October
2021	1	Spider	May
2021	2	Dune	November
2021	3	Coda	June

Table 16: BCNF Table

In the left table, there are multiple candidate keys: {Title}, {Year, Rank}, {year_mo, Rank}, which means every attribute is part of candidate key and there is no non-key attribute; making it a 3NF table. However, in FD {year_mo} `->` {Year}, {year_mo} is not a superkey, making it not a BCNF table. The right table is a BCNF table.

- **Re-define 3NF and BCNF:**
  Sometimes, there is no tables as references, we need to follow these rules to identify if a table is 3ND or BCNF:
  - A database schema S is in **3NF** w.r.t a set of FDs F iff for every relation R in S and for every F in F, it holds that:
    1. F is trivially satisfies. **or**
    2. lhs of F is a superkey of R. **or**
    3. Each attribute in rhs-lhs (in rhs, but not in lhs) of R is part of a candidate key of R.
  - A database schema S is in **BCNF** w.r.t a set of FDs F iff for every relation R in S and for every F in F, it holds that:
    1. F is trivially satisfies. **or**
    2. lhs of F is a superkey of R.

4 Data Storage and Analytical Systems



4.1 RDBMS:

A Relational Database Management System **RDBMS** is a program used to create, update, and manage relational databases, which is the basis for all modern database systems such as MySQL.

4.2 Data Warehouse vs. Data Lake:

A **data warehouse** is a structured repository optimized for storing, querying, and analyzing large volumes of data from various sources, while a **data lake** is a vast, unstructured storage system that holds raw data in its original format until it's needed for processing and analysis.

Granularity of the fact table refers to the level of detail or specificity at which data is stored in the fact table of a data warehouse. Higher (lower) granularity implies more (fewer) rows.
There is a trade-off between level of detailed analysis and storage requirements: finer granularity offers detailed insights but needs more storage (more rows to store).

Aspect	**Data Warehouse**	**Data Lake**
Data Type	Structured, processed data	Raw, unprocessed data (structured, semi-structured, unstructured)
Schema	Schema-on-write (defined before storing data)	Schema-on-read (defined when data is read)
Purpose	Reporting, analysis, and business intelligence	Data exploration, machine learning, and big data analytics
Storage Cost	Typically more expensive due to structured data	Cost-effective for large volumes of raw data
Processing	Optimized for complex SQL queries	Supports multiple processing engines (e.g., Hadoop, Spark)
Users	Business analysts, decision-makers	Data scientists, data engineers
Data Quality	High-quality, cleaned, and curated data	Raw data, may include duplicates or errors

4.3 OLTP vs. OLAP:

Online Transaction Processing **OLTP** is optimized for transactional processing and real-time updates. While Online Analytical processing **OLAP** is optimized for complex data analysis and reporting for business intelligence.

4.4 Multi-dimensional Data aka Data Cube:

A **data cube** is a multidimensional data structure used in **OLAP** to represent data across multiple dimensions, enabling quick and flexible querying and analysis of relationships between different data points.

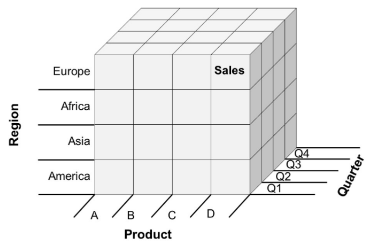


Figure 1: example 3D data cube

This data cube represents sales data across three dimensions: Region, Product, and Quarter. Each cell represents one data point, showing a product being sold at where and when.

PRODUCT	QUARTER	REGION	SALES
A	Q1	Europe	10
A	Q1	America	20
A	Q2	Europe	20
A	Q3	America	50
B	Q4	Europe	60
B	Q2	America	80

Table 17: Corresponding Data Sample (how cube stored)

- **Roll-up:**
  It is the process of summarizing or aggregating data by moving up a different level of granularity with one or more dimensions, such as combining product-specific data into category-level data.

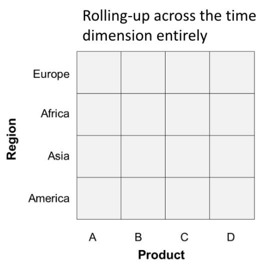


Figure 2: Roll-up example

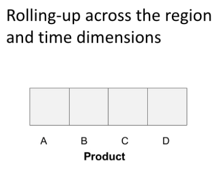


Figure 3: Roll-up example

This left example shows a roll-up operation across the time dimension, where quarters data is completely removed, leaving only the aggregated sales data by region and product.

- **Cross-tab aka pivot-table:**
  It is a table that displays the summarized data across two or more dimensions, allowing for easy comparison and analysis of relationships between the selected dimensions.

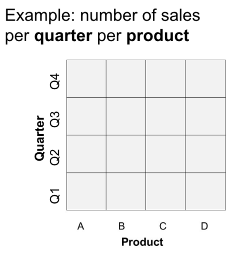


Figure 4: Cross-tab example

The cross-tab displays the number of sales for each product across different time quarters, allowing comparison of sales performance over time for each product.

- **Drill-down:**
  It is the process of retrieving more specific, detailed data by navigating deeper into one or more dimensions, such as breaking aggregated sales data into finer levels like quarters or months.

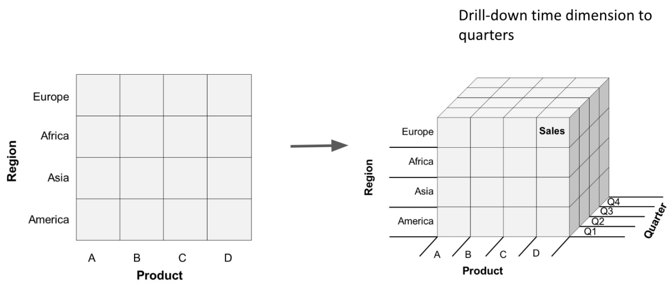


Figure 5: Drill-down example

This example shows a drill-down operation where the time dimension is expended from no time granularity to quarters. We could not drilling quarter down to months as quarter is the minimal level granularity in the data cube.

- **Slicing:**
  It is the process of selecting a specific subset of data by fixing one dimension.

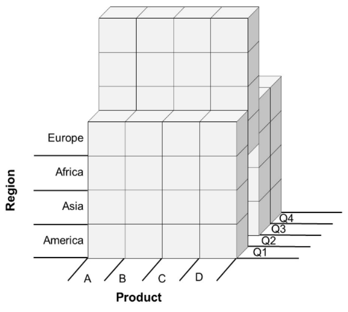


Figure 6: Slicing example

The example represents a slicing operation, where a specific quarter is selected, displaying the sales data across regions and products for that fixed time period.

- **Dicing:**
  It is the process of selecting a subcube by specifying a range of values for multiple dimensions.

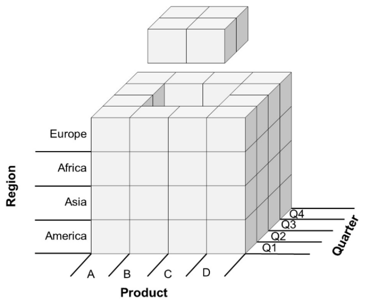


Figure 7: Dicing example

The example represents a dicing operation, where the subcube is selected by choosing specific values or ranges for the region, product, and quarter dimensions.

4.5 Cube Query:

- **Group by Cube:**
  Write a query computes the union of the following groupings of the sales_table being: {(quarter, region), (quarter), (region), ()}, where the () denotes an empty group list (resulting multiset will have: $4 * 2 + 4 * 1 + 2 * 1 + 1 = 15$ tuples):

```sql
SELECT quarter, region, SUM(sales)
FROM sales_table
GROUP BY CUBE (quarter, region);
```

PRODUCT	QUARTER	REGION	SALES
A	Q1	Europe	10
A	Q1	America	20
A	Q2	Europe	20
A	Q3	America	50
B	Q4	Europe	60
B	Q2	America	80

Table 18: sales_table

$\longrightarrow$

QUARTER	REGION	SUM(sales)
Q1	Europe	10
Q1	America	20
Q1	*NULL*	30
Q2	Europe	20
Q2	America	80
Q2	*NULL*	100
Q3	Europe	*NULL*
Q3	America	50
Q3	*NULL*	50
Q4	Europe	60
Q4	America	*NULL*
Q4	*NULL*	60
NULL	Europe	90
NULL	America	150
NULL	*NULL*	240

Table 19: Group by Cube table

- **Group BY Roll-up Query:**
  The first attribute I pass, `quarter`, will be the primary dimension. In this case, we get the total for each quarter and for each region paired with each quarter, but we will not get the total for each region (there will be $4 * 2 + 4 * 1 + 1 = 13$) tuples:

```sql
SELECT quarter, region, SUM(sales)
FROM sales_table
GROUP BY ROLLUP (quarter, region);
```

PRODUCT	QUARTER	REGION	SALES
A	Q1	Europe	10
A	Q1	America	20
A	Q2	Europe	20
A	Q3	America	50
B	Q4	Europe	60
B	Q2	America	80

Table 20: sales_table

$\longrightarrow$

QUARTER	REGION	SUM(sales)
Q1	Europe	10
Q1	America	20
Q1	*NULL*	30
Q2	Europe	20
Q2	America	80
Q2	*NULL*	100
Q3	Europe	*NULL*
Q3	America	50
Q3	*NULL*	50
Q4	Europe	60
Q4	America	*NULL*
Q4	*NULL*	60
NULL	*NULL*	240

Table 21: Group by Roll-up table

- **Grouping Sets:**
  A query computes the union of the following groupings of the `sales_table` being: {(quarter, region), (quarter), (region), ()} (the output will be same as `GROUP BY CUBE` above):

```
        SELECT quarter, region, SUM(sales)
        FROM sales_table
        GROUP BY GROUPING SETS (
            (quarter, region),
            (quarter),
            (region),
            ()
        );
```

4.6 Application Program Interface (API)

An **API** allows one program to share its data with another program or service by providing a standardized way to request and retrieve information. Below is how API works:

1. **Request:** The program sends a request to the application, using a protocol like HTTP or FTP.
2. **Response:** The application responds with the requested information, usually in the form of a JASON object or XML document.

An API is a type of **web service** because it allows apps to interact over the web. To access the API, the program often needs **authentication** using protocols like OAuth to ensure secure data access.

4.7 JavaScript Object Notation (JSON)

JSON is a semi-structured data format that stores data in key/value pairs, where **keys** are unique. Below is an JSON example:

```
{                                   # curly braces enclose a JSoN object
"name": {"first": "Alice",          # keys and values enclosed in double quotes
         "last": "Wang"},
"age": 30,                          # data is separated by commas
"isStudent": false,                 # boolean literals are lowercase
"course": ["Math", "Science"],      # square brackets enclose arrays
"address": {"city": "New-York",     # curly braces enclose a single object
            "zip": "10001"},
"graduationYear": null,             # represented as lowercase
"pets": [{"type": "dog", "name": "woof", "age": 3},
         {"type": "cat", "name": "meow", "age": 10}],
"birth_info": {"date": {"day": 10, "month": "June", "year": 1970},
               "place": {"city": "Oakland", "state": "California"}}
}

# python code for API pull
import urllib, urllib.request
url = '....com'
data = urllib.request.urlopen(url)
print(data.read().decode('utf-8'))   # print jason object in a nice format
```

4.8 MapReduce

MapReduce is a programming model that parallelizes the processing of large datasets across multiple nodes, where each node processes a chunk of data locally. Below is how it works:

1. **Map Phase:** Data is divided into smaller chunks, distributed across nodes. A mapping function is applied to each chunk (eg: counting word occurrences in each document). The result is key-value pairs (eg: "word": count).
2. **Shuffle and Sort:** The key-value pairs are grouped and sorted based on keys (eg: words starting with certain letters are sent to specific intermediate units).
3. **Reduce:** The reduce function aggregates the results fro the map phase (eg: summing word counts from all documents). The final output is a combined result for all chunks (eg: total occurrences of each word across all document).

In the step of Shuffle and Sort, there are several ways to partition data: 1) Range-based: divide data into ranges based on field's value (eg: alphabetically or numerically) and each is assigned to a processor. 2) Hash-based: a key is selected and processor is assigned based on each hash. 3) Round-robin: data is assigned sequentially to processor in a rotating order regardless of its value.

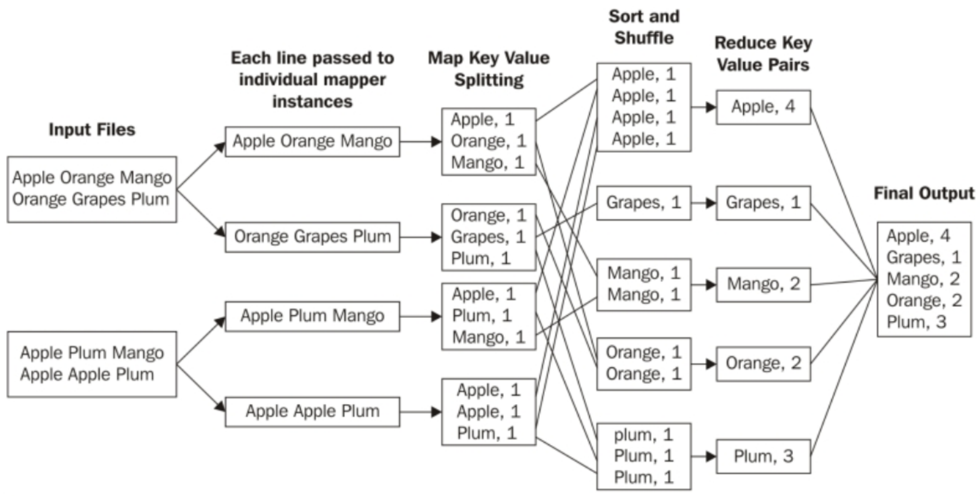


Figure 8: MapReduce example

4.9 Hadoop:

Hadoop is an open source framework based on Java that manages the storage and processing of large amounts of data for applications.

$$\text{Hadoop} = \text{HDFS} + \text{MapReduce}$$

1. **Hadoop Distributed File System (HDFS)** is responsible for storing large amounts of data across multiple nodes in a distributed manner. HDFS breaks a large file into smaller blocks and distributes these blocks across multiple nodes in the cluster. Each block is replicated across several nodes. If one node fails, the data is still accessible from other nodes that store the replicas.

2. **MapReduce** is the processing framework in Hadoop that allows data to be processed in parallel across the cluster.

The core component of the Hadoop is Yet Another Resource Negotiator **YARN**:

1. **ResourceManager:** acts as the central authority that allocates resources across the cluster.

2. **NodeManager:** runs on each node to manage and monitor resources (like CPU and memory) locally.

3. **ApplicationMaster:** created for each application (such as a MapReduce job) to request resources from the ResourceManager and coordinate execution with the NodeManagers.

5 NoSQL (Not Only SQL) Databases:

5.1 Differences between SQL and NoSQL:

1. **SQL** databases are relational, and **NoSQL** databases are non-relational.
2. **SQL** databases use structured query language (SQL) and have a predefined schema. **NoSQL** databases have dynamic schemas for unstructured data.
3. **SQL** databases are vertically scalable (upgrading hardware to handle more load), while **NoSQL** databases are horizontally scalable (adding more nodes/servers to load large data volumes).
4. **SQL** databases are table-based, while **NoSQL** databases are document, key-value, graph, or wide-column stores.
5. **SQL** databases are better for multi-row transactions, while **NoSQL** is better for unstructured data like documents or JSON.
6. **SQL** requires joins to combine data from multiple tables. **NoSQL**, data is embedded in a single structure (distributes data across multiple servers, each responsible for a shard, optimized for handling high-velocity, high-volume, and high-variety of data).
7. **SQL**, data is normalized to reduce redundancy. While **NoSQL**, data is denormalized to improve query performance.

5.2 Consistency vs. Availability Trade-off:

1. **Consistency:** Consistency means that every replica of a shard must have the exact same data at all times.
2. **Availability:** Availability means that every piece of information in the database is always accessible for both reading and writing, even during failures. For instance, even if some nodes in the database fail, the system should still be able to process read and write request.
3. **Partition Tolerance:** Partition Tolerance means that the cluster must continue to work despite any number of communication breakdowns between shards (nodes) in the system.

CAP Theorem:

1. **Choosing AP (Cassandra):** AP systems prioritize <u>Availability</u> and <u>Partition Tolerance</u>, but sacrifice <u>Consistency</u>. The system guarantees that you can always write or read data, but the data might not be the same across all replicas. It relies on <u>eventual consistency</u>, meaning that once the partition is resolved, all replicas will eventually converge to a consistent state.
2. **Choosing CP (MongoDB):** CP systems emphasize <u>Consistency</u> and <u>Partition Tolerance</u> but compromise <u>Availability</u>. The system will remain consistent, but some operations may be unavailable during a partition.

5.3 MongoDB:

MongoDB uses a replica set architecture. A replica set consists of a primary node and multiple secondary nodes. The primary node handles all write operations, while the secondary nodes replicate the data asynchronously from the primary. If the primary node fails, the secondary nodes detect the failure and select a secondary node as a new primary node. Once the failure is fixed, it rejoins as a secondary and synchronizes with the current primary to maintain data consistency.

RDBMS	**MongoDB**
Database	Database
Relation (Table)	Collection
Row (Record)	Document
Column (Attribute)	Field

- **Document Structure:**
  1. **BSON, or Binary JSON** is the data format that MongoDB uses to organize and store data.
  2. `_id` field acts as the **primary key**, which is indexed by default. If it is not specified during document insertion, MongoDB will generate one automatically.

- **Namespace:**
  Switching to `tutorial` sets the namespace you are working in, which in MongoDB refers to the current database. All operations, like inserting documents or querying data, will then apply to this namespace.

```
use tutorial

show dbs
show collections
```

- **Querying in Mongo:**
  Create (if not exists)/insert a collection called students:

```
# optional
db.createCollection('students')

db.students.insert({name: "Jane Doe",
                    age: 45,
                    hobbies: ['swimming', 'kayaking', 'reading']})
# equivalent to use insertOne()

db.students.countDocuments()   # find how many documents
# Count the number of documents in students collection where name is John:
db.students.countDocuments({name: "John"})
```

Retrieve documents from a collection:

```
# retrieve documents:
db.students.find()

# retrieve documents, name is "Johnny":
db.students.find({name: "Johnny"})

# retrieve the 1st document:
db.students.findOne()

# retrieve documents with name is "Johnny", include name, age fields:
db.students.find({name: "Johnny"}, {name: 1, age: 1})

# retrieve documents with name is "Johnny", excluding age field:
db.students.find({name: "Johnny"}, {age: 0})
```

Retrieve documents where age is greater than or equal to 30:

```
db.students.find({age: {'$gte': 30}})

# strictly greater than: $gt
# strictly less than: $lt
# less than or equal to: $lte
```

Retrieve documents where name is `Johnny` **and** tot_cred is less than 8:

```
db.students.find({$and: [name: "Johnny", tot_cred: {$lt: 8}]})
```

Retrieve documents where name is `Johnny` and either tot_cred is less than 100 **or** dept is `DSI`:

```
db.students.find({name: "Johnny", $or: [{tot_cred: {$lt: 100}},
                                        {dept: "DSI"}]})
```

Retrieve documents where the value of name if equal to the value of age using expression:

```
db.students.find({'$expr': {$eq: ['$name', '$age']}})
```

Insert multiple documents into a collection in a single operation:

```
db.students.insertMany([{name: "Jack Dorsey", age: 56, hobbies: ['tech']},
                        {name: "John Smith", age: 23}])
```

Update documents:

```
# add a new field, country:
db.students.update({name: "John-Doe"}, {$set: {country: 'USA'}})

# change the original document, name:
db.students.update({name: "John-Doe"}, {$set: {name: "Johnny"}})

# replace {name: "Johnny"} with {status: "new"}:
db.students.update({name: "Johnny"}, {status: "new"})
```

Delete documents:

```
# deleting documents without deleting collection:
db.students.remove()

# deleting a specific document:
db.students.remove({name: "Johnny"})

# deleting collection:
db.students.drop()

# delete documents from students:
db.students.deleteMany({})
```

5.4 PyMongo:

PyMongo is a Python library used to interact with MongoDB.

- **Mongo Client:**
  A MongoDB server is needed with PyMongo to store, manage, and handle database operations, while PyMongo acts as the client that communicates with the server to perform these tasks.

```
pip install pymongo
import pymongo
from pymongo import MongoClient

client = MongoClient("localhost", 27017)
```

- **Namespace and Collections:**

```
# list all database names:
print(client.list_database_names())

# access a specific database named "students":
db = client["student"]
# euqivalent:
db = client.students

# list all collection names within the "students" database:
print(db.list_collection_names())

# access a specific collection named "myCollection":
collection = db["myCollection"]
# equivalent:
collection = db.myCollection
```

- **Querying in PyMongo:** Retrieve the first document from `prizes` collection where the `category` field is `chemistry`:

```
document = db.prizes.find_one({"category": "chemistry"})
print(document)
```

Retrieve documents where the `category` is `chemistry` **and** the `year` is 2020:

```
document = db.prizes.find({"category": "chemistry", "year": 2020})
print(document)
```

Retrieve documents where the `category` is `chemistry` **or** the `year` is 2020:

```
    documents = db.prizes.find({
        "$or": [
            {"category": "chemistry"},
            {"year": 2020}
        ]
    })

    for x in document:
        print(x)
```

Retrieve documents where the `year` is greater than 2018:

```
    documents = db. prizes.find({
        "year": {"$gt": 2018}
    })

    for x in documents:
        print(x)
```

Retrieve documents where the `category` is `peace`, **include** `category`, `year`, `firstname`, and `lastname`, **excluding** `_id`. Then **sort** by `year` in **ascending** order and `category` in **descending** order. **Limit** the results to 2 documents:

```
    results = db.prizes.find(
        {"category": "peace"},
        {
            "_id": 0, "category": 1, "year": 1,
            "firstname": 1, "lastname": 1
        }
    ).sort([("year", 1), ("category", -1)]).limit(2)

    for x in results:
        print(x)
```

6 Spark:

Apache Spark is a fast, in-memory data processing framework. Unlike Hadoop's batch processing, Spark's in-memory operations make it faster and more efficient for iterative and interactive tasks.
In Apache Spark, transformations define what to do with data, but the computation doesn't occur until an action is called.

RDD → DataFrame → Dataset

1. **Resilient Distributed Dataset (RDD):** RDD is the foundational Spark API for distributed data, which supports low-level transformation (`map`, `filter`) with fault tolerance.
2. **DataFrame:** Built on top of RDDs, DataFrames represent data in a structured, tabular format with columns, which is optimized for SQL-like syntax and expression-based operations.

```
        # converting between RDDs and DF
        df = rdd.toDF()
```

3. **Dataset:** Datasets combine the optimization of DataFrames with the type safety of RDDs, which provides compile-time error checking.

6.1 Spark Session:

Set up:

```
from pyspark.sql import SparkSession

# creating a spark_session:
spark = SparkSession \
    .builder \
    .getOrCreate()
    # .appName("MyAppName") \
    # .config("spark.some.config.option", "some-value") \

df = spark.read.csv("path/to/file.csv", header = True)
df = spark.read.json("path/to/file.json")
df.show()
```

Converts between Spark DataFrame and Pandas DataFrame:

```
pandas_df = df.toPandas()

spark_df = sparkCreateDataframe(pandas_df)
```

JSON collections can be turned into a Spark DataFrame, we can also turn a Spark DataFrame into a collection of JSON documents.

```
df = spark.createDataFrame([(2, 'Alice'), (5, 'Bob')], schema = ['age', 'name'])

df.toJSON()
```

EDA to access meta-data:

```
df.printSchema()
df.count()
df.describe('column_name').show()
df.select('column_name').summary().show()
df.drop('column_name')
```

Group by `birthYear`, counts the occurrences of each birth year, sorts them in descending order of count, and displays the result:

```
df.groupBy("brithYear").count().orderBy("count", ascending = False).show()
```

Join two DataFrames:

```
joined_df = df1.join(df2, on = "column", how = "inner/left")

# cross product
cross_df = df1.crossJoin(df2)
```

Append a DataFrame `df2` to `df1`, the rows of `df2` will appear after the rows of `df1`. They must have the same number of columns:

```
appended_df = df1.union(df2)
```

Computes a pair-wise frequency table of the given columns (contingency table):

```
df.crosstab("c1", "c2").show()
```

c1	c2
1	11
1	11
3	10
4	8
4	8

$\longrightarrow$

c1_c2	10	11	8
1	0	2	0
3	1	0	0
4	0	0	2

Add a column called `age_plus_2`:

```
df.withColumn('age_plus_2', df.age+2).show()
```

Calculate the count of records for each combination of `company` and `status`, including all possible aggregations (same as groupby Cube):

```
df.cube('company', 'status').count().show()
```

status	company
LLC	XYZ
INC	ABC

$\longrightarrow$

company	status	count
XYZ	LLC	1
XYZ	NULL	1
ABC	NULL	1
ABC	INC	1
NULL	LLC	1
NULL	INC	1
NULL	NULL	2

Similarly, group by Roll Up:

```
df.rollup('company', df.status).count().show()
```

status	company
LLC	XYZ
INC	ABC

$\longrightarrow$

company	status	count
XYZ	LLC	1
XYZ	NULL	1
ABC	NULL	1
ABC	INC	1
NULL	NULL	2

Statistical analysis:

```
# Pearson Correlation:
df.corr('c1', 'c2')    # c1 and c2 are features in df

# Covariance:
df.cov('c1', 'c2')
```

Create an intersection of two DataFrames (same as Intersect in SQL):

```
df1.intersect(df2)
```

User Defined Functions (UDFs): Define a UDF to convert text to uppercase, applies it to the `name` column to create a new column called `Capitalized` in the DataFrame, and displays the result without truncating:

```
upperCaseUDF = udf(lambda z: upperCase(z))
df.withColumn('Capitalized', upperCaseUDF(col('name'))).show(truncate = False)
```