Week 3: Intermediate and Advanced SQL

L3.1: *SQL Examples*

- This lecture just consists examples of all the topics we have learned in **Week 2**.
- You can check the SQL Example video here 🔗

L3.2: *Intermediate SQL - Part 1*

Nested Subqueries

- We can write a query (sub-query) inside a query, called a **nested subquery**.
- A **subquery** is a `SELECT-FROM-WHERE` expression that is nested within another query.
- The nesting can be done in the following SQL query:

$$\begin{array}{rl} \text{SELECT} & A_1, A_2, \dots, A_n \\ \text{FROM} & r_1, r_2, \dots, r_m \\ \text{WHERE} & B < operation > (subquery) \end{array}$$

  - A_i can be an attribute or an expression.
  - r_i can be a relation or a subquery.
  - B is an attribute and operation to be performed on it.

Set Membership

- Set membership is used to check whether a particular tuple(row) is a member of a relation(table) or not.
- This can be done using `IN` and `NOT IN` operators.

SYNTAX:

```
SELECT column1, column2
FROM table_name
WHERE column1 IN (value1, value2, value3);
```

Example:

- Find the names of all the students who have taken a course in the **Sep 2022** and **Jan 2023 term**.

```
SELECT distinct name
FROM stduents
WHERE term="Sep" and year=2022
AND id IN (
    SELECT id
    FROM students
    WHERE term="Jan" and year=2023
);
```

Set Comparison

`SOME` Clause

- The `SOME` clause will give the result if the condition is true for **at least one** of the tuples in the subquery.

SYNTAX:

```
SELECT column1, column2
FROM table_name
WHERE column1 > SOME (SELECT column1 FROM other_table);
```

Example:

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

  ```
  SELECT name
  FROM instructor
  WHERE salary > SOME (
      SELECT salary
      FROM instructor
      WHERE dept_name="Biology"
  );
  ```

`ALL` Clause

- The `ALL` clause will give the result if the condition is true for **all** of the tuples in the subquery.

SYNTAX:

```
SELECT column1
FROM table_name
WHERE column1 > ALL (SELECT column1 FROM other_table);
```

Example:

- Find the names of instructors with salary greater than that of all instructors in the Biology department

```sql
SELECT name
FROM instructor
WHERE salary > ALL (
    SELECT salary
    FROM instructor
    WHERE dept_name="Biology"
);
```

`EXISTS` Clause

- The `EXISTS` clause will return `True` if the subquery returns at least one tuple, else it will return `False`.
- If it returns `True`, outer query retrives the specified columns from the sub query.

SYNTAX:

```sql
SELECT column1, column2, ...
FROM table_name
WHERE EXISTS (subquery);
```

Example:

- Find all courses taught in both the **Fall 2009** and **Spring 2010** terms.

```sql
SELECT course_id
FROM section AS S1
WHERE semester="Fall" AND year=2009
AND EXISTS (
    SELECT *
    FROM section AS S2
    WHERE semester="Spring" AND year=2010
    AND S1.course_id = S2.course_id
);
```

`NOT EXISTS` Clause

- The `NOT EXISTS` clause will return `True` if the subquery returns no tuples, else it will return `False`, it's the opposite of `EXISTS` clause.
- If it returns `True`, outer query retrives the specified columns from the sub query.

SYNTAX:

```sql
SELECT column1, column2, ...
FROM table_name
WHERE NOT EXISTS (subquery);
```

Example:

- Find all customers who do not have any orders.

```sql
SELECT *
FROM customers
WHERE NOT EXISTS (
    SELECT *
    FROM orders
    Where orders.customer_id = customers.customer_id
);
```

Subqueries in `FROM` Clause

- SQL allows a subquery expression to be used in the `FROM` clause.

SYNTAX:

```sql
SELECT col1, ...
FROM (
    SELECT attr1, ...
    FROM tableX, ...
    WHERE condition
) AS table1
WHERE condition;
```

Example:

- Find the average instructors salaries of those deparatments whose average salary is greater than $40,000$.

```sql
SELECT dept_name, avg_salary
FROM (
    SELECT dept_name, avg(salary) AS avg_salary
    FROM instructor
    GROUP BY dept_name
)
WHERE avg_salary > 40000;
```

`WITH` Clause

- The `WITH` clause is used to define a temporary relation (table) whose definition is available only to the query in which the `WITH` clause occurs.

SYNTAX:

```sql
WITH temp_table_name(attribute_list) AS (
    SELECT exp1, ...
    FROM table1, ...
    WHERE condition
)
SELECT temp_table_name.attribute1
FROM temp_table_name, ...
WHERE condition;
```

Example:

- Find all departments with the maximum budget

```
WITH max_budget(budget) AS (
    SELECT max(budget)
    FROM department
)
SELECT dept_name
FROM department, max_budget
WHERE department.budget = max_budget.budget;
```

- Find all orders having more than average total amount

```
WITH avg_total(total) AS (
    SELECT avg(total_amount)
    FROM orders
)
SELECT *
FROM orders, avg_total
WHERE orders.total_amount > avg_total.total;
```

Scalar Subquery

- Scalar Subquery is used where a single value is expected
- It gives *Runtime* Error if the subquery returns more than one tuple.

SYNTAX:

```
SELECT <attribute>, (
    SELECT scalar_subquery
) AS <alias>
FROM <relation>;
```

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;
```

`CASE-WHEN` Expression

- The `CASE` statement is used to evaluation a condition and return a value Yased on the outcome.
- The `ELSE` clause is used to specify a default value that will be returned if none of the conditions are met.

SYNTAX:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE result
END
```

Example:

- Give discount in total bill of 10% where bill is equal to or greater than $5,000$, and 5% where bill is less than $5,000$.

  ```
  SELECT order_id, (
      CASE
          WHEN total_amount >= 5000 THEN total_amount * 0.9
          ELSE total_amount * 0.95
      END
  ) AS total_after_discount
  FROM orders;
  ```

L3.3: *Intermediate SQL - Part 2*

Join Expressions

- Join operations take two relations and returns as a result another relation.
- A join operation is a Cartesian product followed by a selection.
- The join operations are typically used as a subquery expressions in the `FROM` clause.

Example we will be using:

$Course$

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

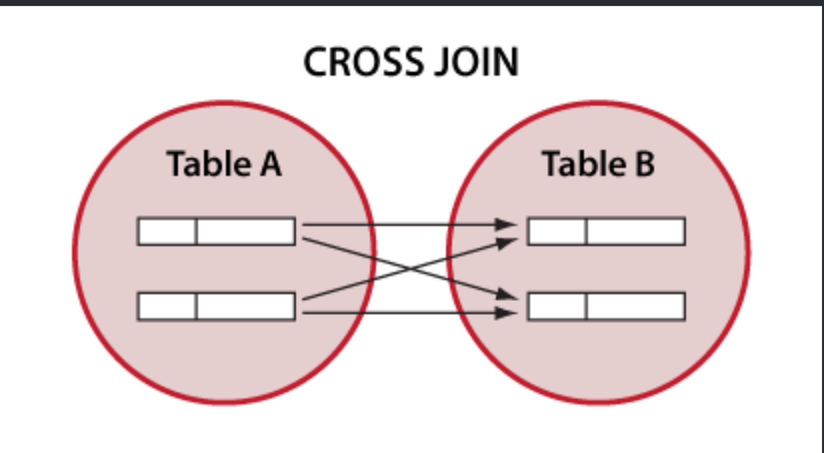
$prereq$

course_id	prereq_id
BIO-301	BIO-101

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

Cross Join

- `CROSS JOIN` returns the Cartesian product of the two relations.



Explicit Syntax:

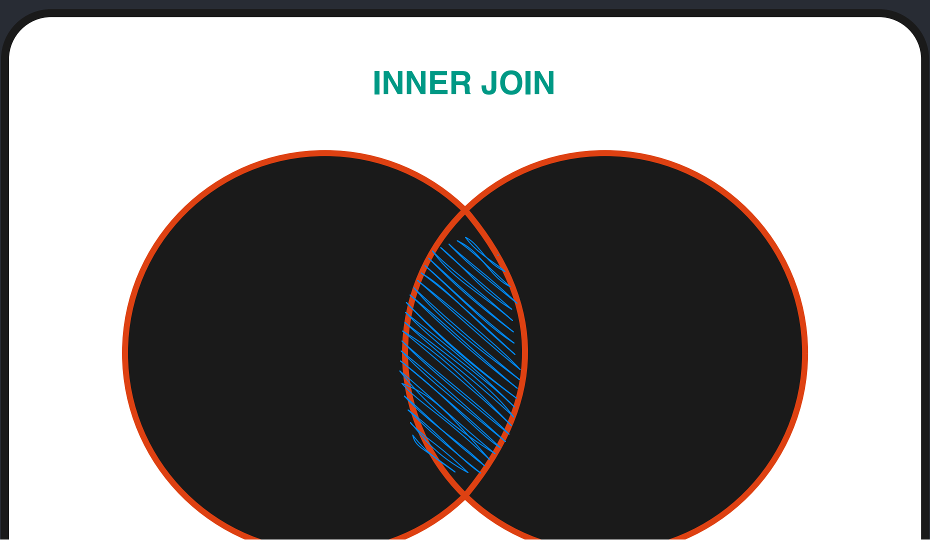
```
SELECT *
FROM table1 CROSS JOIN table2;
```

Implicit Syntax:

```
SELECT *
FROM table1, table2;
```

Inner Join

- `INNER JOIN` returns the tuples from all the relations.
- It includes same column names from both the tables on which the join is performed.
- It includes only those tuples which satisfy the join condition.



SYNTAX:

```sql
SELECT columns
FROM table1
INNER JOIN table2 ON table1.column = table2.column
INNER JOIN table3 ON table2.column = table3.column;
```

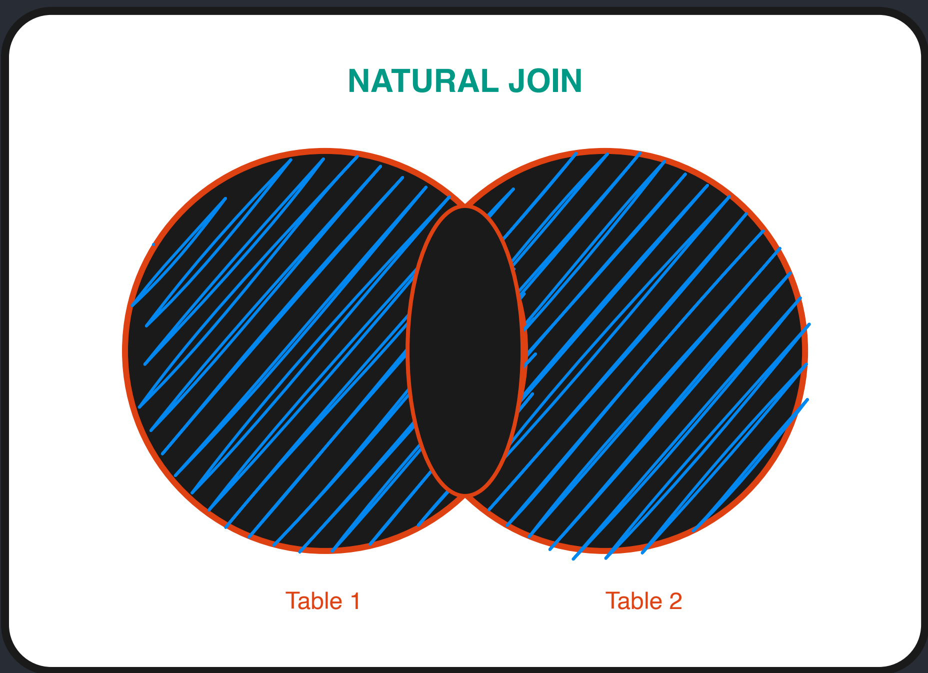
Example:

```sql
SELECT *
FROM Course
INNER JOIN prereq ON Course.course_id = prereq.course_id;
```

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

Natural Join

- `NATURAL JOIN` returns the tuples from all the relations.
- It only includes one column name from which the join is performed.
- It only includes those tuples which satisfy the join condition.



SYNTAX:

```sql
SELECT columns
FROM table1
NATURAL JOIN table2;
```

Example:

```sql
SELECT *
FROM Course
NATURAL JOIN prereq;
```

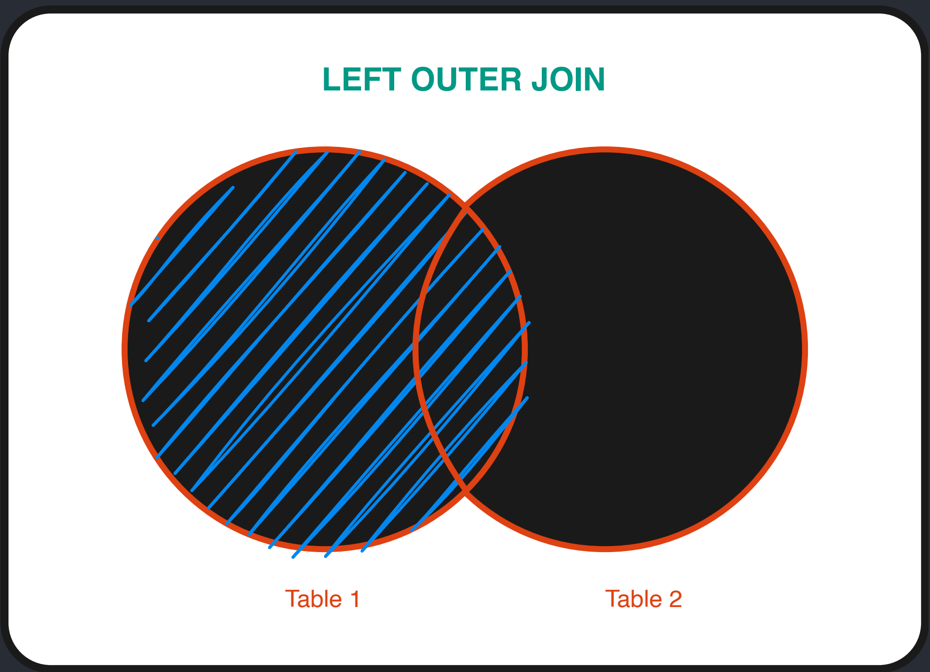
course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

Outer Join

- `OUTER JOIN` returns the tuples from all the relations.
- It includes unmatched tuples from one or both the tables on which the join is performed.
- If there are no mathcing rows, `NULL` values are used.

`LEFT OUTER JOIN`

- `LEFT OUTER JOIN` returns all the tuples from the left table and only those tuples from the right table which satisfy the join condition.
- If there are no matching rows, `NULL` values are used.



SYNTAX:

```sql
SELECT columns
FROM table1
LEFT OUTER JOIN table2 ON table1.column = table2.column;
```

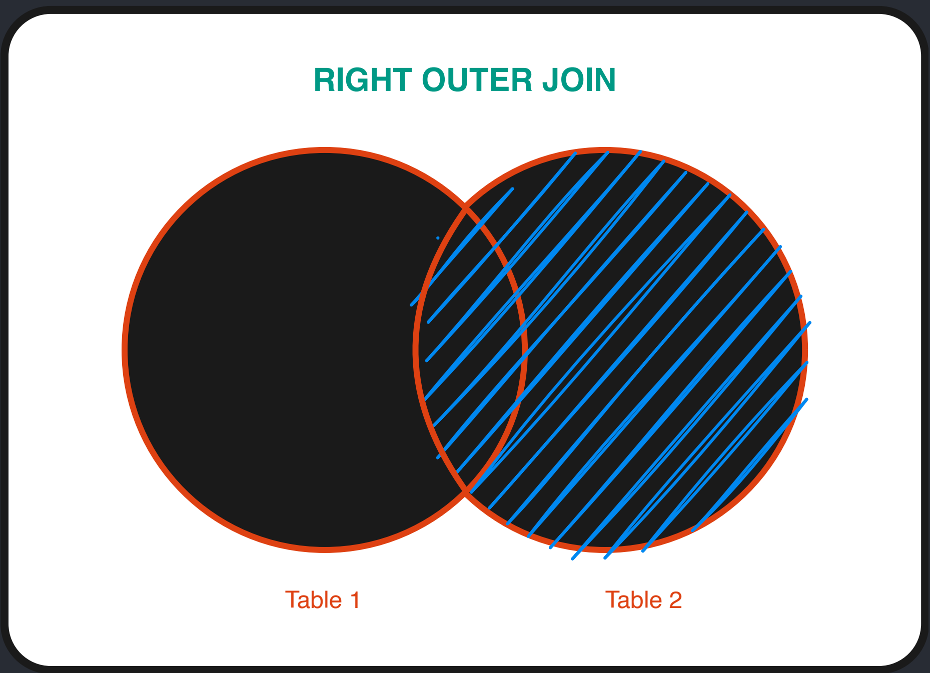
Example:

```sql
SELECT *
FROM Course
LEFT OUTER JOIN prereq ON Course.course_id = prereq.course_id;
```

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

`RIGHT OUTER JOIN`

- `RIGHT OUTER JOIN` returns all the tuples from the right table and only those tuples from the left table which satisfy the join condition.
- If there are no matching rows, `NULL` values are used.



SYNTAX:

```sql
SELECT columns
FROM table1
RIGHT OUTER JOIN table2 ON table1.column = table2.column;
```

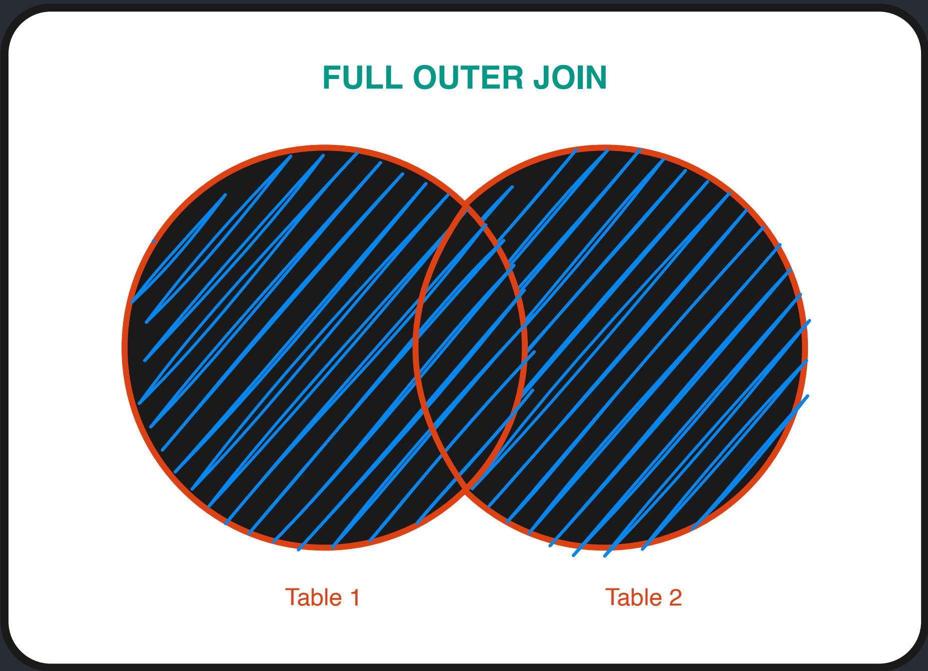
Example:

```sql
SELECT *
FROM Course
RIGHT OUTER JOIN prereq ON Course.course_id = prereq.course_id;
```

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
NULL	NULL	NULL	NULL	CS-347

`FULL OUTER JOIN`

- `FULL OUTER JOIN` returns all the tuples from both the tables.
- If there are no matching rows, `NULL` values are used.



SYNTAX:

```sql
SELECT columns
FROM table1
FULL OUTER JOIN table2 on table1.column = table2.column;
```

Example:

```sql
SELECT *
FROM Course
FULL OUTER JOIN prereq ON Course.course_id = prereq.course_id;
```

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101

course_id	title	dept_name	credits	prereq_id
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	NULL
NULL	NULL	NULL	NULL	CS-347

Views

- *A View is a **Virutal table*** derived from one or more underlying tables or views.

Purpose of Views

- **Simplify complex queries**
  Views can encapsulate complex queries, making them easier to use and understand by providing a simplified and tailored view of the data.
- **Data security**
  Views can restrict access to certain columns or rows of the underlying tables, allowing you to control the level of data visibility for different users.
- **Data Abstraction**
  Views can provide an abstraction layer by presenting a logical representation of the data, hiding the underlying structure and complexity of the data.

SYNTAX

```
CREATE VIEW view_name AS
(
  SELECT column_list
  FROM table_name
  WHERE condition
);
```

Example

- A view of instructors without their salary

```
CREATE VIEW faculty AS
(
    SELECT ID, name, dept_name
    FROM instructor
);
```

- A view of department salary totals, max, min, and average

```sql
CREATE VIEW dept_analysis(
    dept_name, total_salary, max_salary, min_salary, avg_salary
    ) AS (
        SELECT dept_name, SUM(salary), MAX(salary), MIN(salary), AVG(salary)
        FROM instructor
        GROUP BY dept_name
    );
```

Views Defined using Other views

```sql
CREATE VIEW CS_FALL_2020 AS
(
    SELECT course.coruse_id, sec_id, building, room_number
    FROM course, section
    WHERE course.course_id = section.course_id
    AND course.dept_name = "CS"
    AND section.semester = "Fall"
    AND section.year = 2020
);
```

- The above view can be used to create another view

  ```sql
  CREATE VIEW CS_FALL_2020_BRAHMA AS
  (
      SELECT course_id, room_number
      FROM CS_FALL_2020
      WHERE building = "Brahma"
  );
  ```

Types of Views

- **Depend directly**

  A view is said to depend directly on a table if it is created using the `FROM` clause and the table is listed in the `FROM` clause.

  - **Example**: The following view depends directly on the `customers` table:

    ```sql
    CREATE VIEW customers_with_high_balance AS
    (
        SELECT customer_id, name, balance
        FROM customers
        WHERE balance > 10000
    );
    ```

- **Depend on**

  A view is said to depend on a table if it is craeted using the `FROM` clause and the table is indirectly referenced by another table that is listed in the `FROM` clause.

  - **Example**: The following view depends on the `orders` table because the `orders` table is indirectly referenced by the `products` table.

```sql
CREATE VIEW products_ordered_by_customers AS
(
product_id, name, quantity
FROM products
JOIN orders ON orders.product_id = products.product_id
WHERE order_status = 'Completed'
);
```

- **Recursive views**

  A recursive view is a view that references itself in its `FROM` clause.

  Recursive views can be used to create self-joins or to traverse a hierarchical data structure.

  - **Example**: The following recursive view will return all customers and their children:

    ```sql
    CREATE VIEW customers_and_children AS
    (
    SELECT customer_id, name,
        (
            SELECT customer_id, name
            FROM customers
            WHERE customers.parent_id = customer_id
        ) AS children
    FROM customers
    WHERE parent_id IS NULL
    );
    ```

View Expansion

- It refers to the process by which a query involving a view is rewritten or expanded into an equivalent query that directly accesses the underlying tables.
- When a query is executed against a view, the database system may interanlly perform view expansion to optimize the query execution.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  ```
  REPEAT
      Find any view relation vi in e1
      Replce the view realtion vi by the expression defining vi
  UNTIL no more view relations are present in e1
  ```

- As long as the view definitions are not recursive, this loop with terminate.

Updation of a View

- Adding a new tuple to some view v_1 lead to updation in the underlying base tables.

Example:

- Adding a new tuple to *faculty* view

```sql
INSERT INTO faculty
VALUES ('30123', 'Orange', 'Music');
```

- This insertion will represent insertion of this tuple in underlying *instructor* table

```sql
INSERT INTO instructor
VALUES ('30123', 'Orange', 'Music', null);
```

Materialized Views

- **Materializing a view** will create a physical table containing all the tuples in the result of the query defining the view
- If relations used in the query are updated, the maerialized view will not be updated automatically.

L3.4: *Intermediate SQL - Part 3*

Transactions

- a Transaction is a group of one or more SQL statements that are treated as a single unit of work.
- If the transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database.
- If the transaction encounters ERRORS and must be cancelled or rolled back, then all of the data modifications are erased.
- Transactions commands comes under **Transaction Control Language (TCL)**

ACID Properties

- Transactions are used to ensure the **ACID** properties of data

Atomicity

- A transaction is an atomic unit of work.
- This means that either all of the statements in the transaction are executed successfully, or none of them are executed.

Consistency

- A transaction must maintain the consistency of the database.
- This means that the database must be in a valid state after the transaction is committed.

Isolation

- Transactions must be isolated from each other.
- This means that the changes made by one transaction must not be visible to other transactions until the first

transaction has been committed.

Durability

- Once a transaction has been committed, the changes made by the transaction must be permanent.
- This means that they must not be lost even if the database crashes or the server loses power.

> To start a transaction, we use the `BEGIN TRANSACTION` statement.
>
> To commit a transaction, we use the `COMMIT` statement.
>
> To rollback a transaction, we use the `ROLLBACK` statement.

Integrity Constraints

- **Integrity constraints** are rules or conditions that are defined on database tables to enforce data integrity and maintain the consistency of data.
- Integrity constraints commands comes under **Data Definition Language (DDL)**.

`NOT NULL`

- The `NOT NULL` constraint ensures that a column cannot have a NULL value.

```
CREATE TABLE customers(
    customer_id INT,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(255) NOT NULL
);
```

`PRIMARY KEY`

- The `PRIMARY KEY` constraint uniquely identifies each record in a table.

```
CREATE TABLE customers(
    customer_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(255) NOT NULL
);
```

`UNIQUE`

- The `UNIQUE` constraint ensures that all values in a column are different.

```
CREATE TABLE customers(
    customer_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE
);
```

- It will make the column a candidate key.
- Candidate keys are permitted to be null.

`CHECK(P)`, where P is a predicate

- The `CHECK` constraint ensures that all values in a column satisfy a specific condition.

```
CREATE TABLE customers (
  customer_id INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  balance INT CHECK (balance >= 0),
  PRIMARY KEY (customer_id)
);
```

Referential Integrity

- It ensures that if a value of one attribute of a relation (table) references a value of another attribute on another relation (table), then the referenced value must exist.

There are various types of referential integrity constraints, some are:

`ON DELETE CASCADE`

- This constraint specifies that if a row is deleted from the parent table, then all rows in child table that reference the deleted row will also be deleted.

```
CREATE TABLE customers (
  customer_id INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  PRIMARY KEY (customer_id)
);

CREATE TABLE orders (
  order_id INT NOT NULL AUTO_INCREMENT,
  customer_id INT NOT NULL,
  order_date DATETIME NOT NULL,
  total_amount INT,
  PRIMARY KEY (order_id),
  FOREIGN KEY (customer_id) REFERENCES customers (customer_id)
  ON DELETE CASCADE
);
```

`ON UPDATE CASCADE`

- This constraint specifies that if a row is updated in the parent table, then all rows in child table that reference the updated row will also be updated.

```sql
CREATE TABLE customers (
  customer_id INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  PRIMARY KEY (customer_id)
);

CREATE TABLE orders (
  order_id INT NOT NULL AUTO_INCREMENT,
  customer_id INT NOT NULL,
  order_date DATETIME NOT NULL,
  total_amount INT,
  PRIMARY KEY (order_id),
  FOREIGN KEY (customer_id) REFERENCES customers (customer_id)
  ON UPDATE CASCADE
);
```

`ON DELETE SET NULL`

- This constraint specifies that if a row is deleted from the parent table, then all rows in child table that reference the deleted row will have the referencing column set to `NULL`.

```sql
CREATE TABLE products (
  product_id INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  PRIMARY KEY (product_id)
);

CREATE TABLE orders (
  order_id INT NOT NULL AUTO_INCREMENT,
  product_id INT NOT NULL,
  order_date DATETIME NOT NULL,
  total_amount INT,
  PRIMARY KEY (order_id),
  FOREIGN KEY (product_id) REFERENCES products (product_id)
  ON DELETE SET NULL
);
```

> Alternative actions to cascade:
>
> - `NO ACTION`
> - `SET DEFAULT`
> - `RESTRICT`

Built-in Data Types

`DATE`

- The `DATE` type is used for values with a date part but no time part.
- It's in the format: `YYYY-MM-DD`

- Example: `DATE '2023-06-30'`

`TIME`

- The `TIME` type is used for values with a time part.
- It's in the format: `HH:MI:SS:MS`
- Example: `TIME '12:30:45'` or `TIME '12:30:45.56'`

`TIMESTAMP`

- The `TIMESTAMP` type is used for values that contain both date and time parts.
- It's in the format: `YYYY-MM-DD HH:MI:SS:MS`
- Example: `TIMESTAMP '2023-06-30 12:30:45.56'`

`INTERVAL`

- The `INTERVAL` type is used for values that is a period of time.
- Example: `INTERVAL '1' DAY` or `INTERVAL '1' YEAR`
- Subtracting a date/time/timestamp value from another gives an interval value
- Interval values can be added to date/time/timestamp values

Index

- An index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.

Example

```sql
CREATE TABLE customers (
  customer_id INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  PRIMARY KEY (customer_id),
);
CREATE INDEX idx_name ON customers (email);
```

- Index uses a B-tree data structure to store data.

User-Defined Data Types

- We can create our own data types using the `CREATE TYPE` statement.

```
CREATE TYPE student_details AS (
    name VARCHAR(255),
    major VARCHAR(255),
)

CREATE TABLE students (
    student_id INT NOT NULL AUTO_INCREMENT,
    details student_details,
    PRIMARY KEY (student_id)
);
```

Example table:

student_id	details
1	(John Doe, Computer Science)
2	(Jane Doe, Accounting)
3	(Peter Smith, Engineering)

Domains

- A domain is a user-defined data type that has a set of valid values.

Example:

```
CREATE domain degree_level VARCHAR(50) NOT NULL
CONSTRAINT degree_level_test CHECK (value IN ('Foundation', 'Diploma', 'Degree'));
```

Large Objects Types

`BLOB`

- The `BLOB` type is used for storing large binary objects.
- Example: Storing images, audio, video, etc.

`CLOB`

- The `CLOB` type is used for storing large text objects.
- Example: Storing documents, HTML pages, XML data, etc.

> When a query returns a large object, a pointer is returned rather than the large object itself.

Authorization

- Authorization is the process of determining whether a user has permission to access a database object.
- Authorization commands comes under **Data Control Language (DCL)**.
- Forms of authorization on parts of the database:
  - `Read` (`SELECT`) - Allow reading of data.
  - `Insert` - Allow insertion of new data, not modification of existing data.
  - `Update` - Allow modification of existing data.
  - `Delete` - Allow deletion of data.
  - `all privileges` - Allow all of the above.
- Forms of authorization to modify the database schema
  - `Index` - Allow creation and deletion of indices
  - `Resources` - Allow createion of new relations
  - `Alteration` - Allow addition or deletion of attributes in a relation
  - `Drop` - Allow deletion of relations

Authorization Specification

- Authorization is specified using the `GRANT` and `REVOKE` statements.

```
GRANT privileges ON relation/view_name TO user_list;
```

```
REVOKE privileges ON relation/view_name FROM user_list;
```

- `user_list` is a user-id, public, or a role.
- Granting a privilege on a view does not imply granting any privileges on underlying relations.

Roles

- A role is a named group of privileges.
- Roles can be granted to users.
- Creating a role

  ```
  CREATE ROLE role_name;
  ```

  - Example:

    ```
    CREATE ROLE student;
    ```

- Granting a role to a user

  ```
  GRANT role_name TO user_list;
  ```

  - Example:

    ```
    GRANT student to Param;
    ```

- Granting privilege to roles:

```
GRANT privilege ON relation/view_name TO role_name;
```

  - Example:

  ```
  GRANT SELECT ON students_records TO student;
  ```

- Roles can be granted to other roles.

```
GRANT role_name TO role_name2;
```

  - Example:

  ```
  GRANT TA TO student;
  ```

L3.5: *Advanced SQL*

Functions and Procedures

- Functions are procedures are created using **Data Definition Language (DDL)**.

Functions

- Functions are used to perform a specific task such as performing calculations, transformations on data.
- Functions returns a value.
- Functions can be called from within other queries.

SYNTAX: Example function

```
CREATE FUNCTION sum2 (a INT, b INT) RETURNS INT
BEGIN
    DECLARE sum INT;
    SET sum = a + b;
    RETURN sum;
END;
```

Calling

```
SELECT sum2(1, 2);
```

Procedures

- Procedures are used to perform actions on data such as inserting, updating, deleting rows from a table.
- Procedures do not return a value.

- Procedures cannot be called from within other queries.
- Procedures support control flow constructs such as conditional statements (`IF-THEN-ELSE`), loops (`WHILE`, `REPEAT` & `FOR`) and exceptional handling (`TRY-CATCH`).

SYNTAX: Example procedure

```
CREATE PROCEDURE my_procedure (IN val1 INT, IN val2 VARCHAR(30))
BEGIN
  INSERT INTO my_table (column1, column2) VALUES (val1, val2);
END;
```

Invoking

```
CALL my_procedure(1, 'Hello');
```

> ### Overloading
>
> - SQL allows overloading of functions and procedures.
> - This means that multiple functions/procedures can have the same name but different parameters.

Most database systems implement their own variant of the standard syntax.

Loops

- SQL supports two types of loops:
  - `WHILE` loop
  - `REPEAT` loop

`WHILE` loop

- The `WHILE` loop executes a block of code repeatedly as long as a condition is true.

SYNTAX: Example WHILE loop

```
WHILE condition DO
    sequence of statements
END WHILE;
```

`REPEAT` loop

- The `REPEAT` loop executes a block of code repeatedly until a condition is true.
- This is similar to a `DO-WHILE` loop in other programming languages.

SYNTAX: Example REPEAT loop

```
REPEAT
    sequence of statements
UNTIL condition
END REPEAT;
```

`FOR` loop

- The `FOR` loop executes a block of code repeatedly for a specified number of times.

SYNTAX: Example `FOR` loop using `IN`

```
FOR variable_name IN sequence_of_values DO
  -- Body of the loop
END LOOP;
```

SYNTAX: Example `FOR` loop using `AS`

```
DECLARE
n INTEGER DEFAULT 0;
FOR r AS
    SELECT budget FROM department
DO
    SET n = n + r.budget;
END FOR;
```

Conditional Statements

- SQL supports two types of conditional statements:
  - `IF-THEN-ELSE` statement
  - `CASE` statement

`IF-THEN-ELSE` statement

- The `IF-THEN-ELSE` statement executes a block of code if a condition is true, otherwise it executes another block of code.
- If none of the conditions are true, then the `ELSE` block is executed.

SYNTAX: Example `IF-THEN-ELSE` statement

```
IF condition THEN
    sequence of statements
ELSEIF condition THEN
    sequence of statements
ELSE
    sequence of statements
END IF;
```

`CASE` statement

- The `CASE` statement executes a block of code based on a condition.
- It is similar to a `switch-case` statement in other programming languages.

SYNTAX: Example `CASE` statement

```
CASE
    WHEN condition THEN
        sequence of statements
    WHEN condition THEN
        sequence of statements
    ELSE
        sequence of statements
END CASE;
```

`SEARCHED CASE` statement

- The `SEARCHED CASE` statement executes a block of code based on a condition.
- It doesn't have condition, rather it matches the expression with certain value.

SYNTAX: Example `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;
```

Exception Handling

- Exception handling is used to handle errors that occur during execution of a program.
- SQL supports exception handling using the `DECLARE EXIT HANDLER` statement.
    - The `EXIT HANDLER` is a block of code that will be executed when an exception occurs / raised.
    - It can be used to perform any cleanup tasks such as rolling back a transaction or closing a connection.
- The `SIGNAL` statement is used to raise an exception.

SYNTAX: Example `DECLARE EXIT HANDLER` statement

```
DECLARE
  custom_exception_name EXCEPTION;
BEGIN
  -- Try to set the parameter value to 100
  v_parameter_value := 100;

  -- If the parameter value is not a number, raise the custom exception
  IF NOT v_parameter_value IS NUMBER THEN
    RAISE custom_exception_name;
  END IF;

  -- If the custom exception is not raised, continue with the rest of the procedure
END;

DECLARE EXIT HANDLER FOR custom_exception_name
BEGIN
  -- Do something when the INVALID_PARAMETER_VALUE exception is raised
END;
```

Example

```
CREATE PROCEDURE divide_numbers (number1 INT, number2 INT)
BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';

  DECLARE EXIT HANDLER FOR divide_by_zero
  BEGIN
    SELECT 'Division by zero error occurred!';
    LEAVE divide_numbers;
  END;

  IF number2 = 0 THEN
    SIGNAL divide_by_zero;
  END IF;

  SELECT number1 / number2;
END;
```

Note: This exmaple I found from the internet, it's not much clear, but I hope you got the idea, how to use it.

External Language Routines

- An external language routine is a **user-defined routine (UDR)** that is written in an external language.
- It has the ability to incorporate code written in external programming languages into SQL statements or stored procedures.
- This feature allows developers to extend the functionality of SQL by leveraging the capabilities of other programming languages.
- The database server supports UDRs written in a variety of external languages, including C, C++, Java, and

Python
- To create an external language routine, we use the `CREATE EXTERNAL ROUTINE` statement.
- The `CREATE EXTERNAL ROUTINE` statement specifies the name of the routine, the external language in which it is written, and the location of the external code file.
- `CALL` statement is used to call an external language routine.

SYNTAX:

```
CREATE EXTERNAL ROUTINE factorial
LANGUAGE C
LIBRARY my_factorial_library
EXTERNAL NAME factorial
```

- This statement creates a UDR named `factorial` that is written in C. The `my_factorial_library` **library** contains the code for the `factorial` routine. The `factorial` routine is called `factorial` in the external code file.
- TO call the `factorial` UDR, we use the `CALL` statement as follows:

```
CALL factorial(5);
```

Triggers

- Ttrigger in SQL is a stored procedure that is automatically executed when a specific event occurs on a table.
- Triggers can bew used to enforce business rules, maintain data integrity, maintain an audit trail of changes and automate certain actions within a database.

SYNTAX:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER } {INSERT | UPDATE | DELETE}
ON table_name
[REFERENCING {OLD | NEW} AS alias_name]
[FOR EACH {ROW | STATEMENT} ]
[WHEN (condition)]
BEGIN
    -- Trigger logic and actions go here
END;
```

`BEFORE` triggers

- Run before an update or insert.
- Values that being updated or inserted can be modified before the database is actually modified.

`BEFORE DELETE` triggers

- Run before a delete.

`AFTER` triggers

- Run after an update, insert or delete.
- We can use triggers that run after an operation, such as:
    - Update data in other tables
    - Check against other data in the table or in other tables
    - Run non-database operations coded in user-defined functions

There are 2 types of Triggers:

Row level triggers

- These are executed whenever a row is affected by the event on which the trigger is defined.
- Example:
    - Suppose we have a table `employee` with columns `id`, `name`, `salary` and `bonus`. An `UPDATE` statement is executed to increase the salary of each employee by 10%. A row level `UPDATE` trigger can be defined to update the `bonus` column of the `employee` table based on the updated salary of the employee.

Statement level triggers

- Statement level triggers perform a single action for all rows affected by a statement instead of executing a separate action for each affected row.
- Use `FOR EACH statement` instead `FOR EACH row`.
- Example:
    - Suppose we want to insert multiple rows into a table `employee` from another table `employee_temp`. A statement level `INSERT` trigger will be executed once for the `INSERT` statement instead of executing a separate trigger for each row inserted into the `employee` table.

Example: Trigger to maintain `credits_earned` value

```sql
CREATE TRIGGER credits_earned
AFTER UPDATE OF grades on takes
REFERENCING NEW ROW as nrow
REFERENnCING 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 total_credit = total_credit + (
        SELECT credits
        FROM course
        WHERE course.course_id = nrow.course_id
    )
    WHERE student.id = nrow.id
END;
```

Tutorial 3.1: Triggers (Case studies)

- They have discussed 3 case studies where triggers can be used.