PATUAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

COURSE CODE CCE-223

SUBMITTED TO:

Dr. Md. Samsuzzaman Sobuz

Professor, Department of Computer and Communication Engineering

Faculty of Computer Science and Engineering

SUBMITTED BY:

Name: Mohammed Sakib Hasan

ID: 2102052

Reg No: 101769

Assignment Topic: Chapter Four Theory

4.1 Consider the following SQL query that seeks to find a list of titles of all courses

taught in Spring 2017 along with the name of the instructor.
select name, title
from *instructor* **natural join** *teaches* **natural join** *section* **natural join** *course*
where *semester* = 'Spring' **and** year = 2017
What is wrong with this query?

Ans:

The main issues with the query are:

1. **Risky `NATURAL JOIN` Usage** – It automatically joins tables based on common column names, which may lead to incorrect joins or missing data.
2. **Potential Missing `teaches` Table** – The provided schema does not define a `teaches` table, which is necessary to link `instructor` and `section`.
3. **Ambiguity in Attribute Names** – Columns like `course_id`, `semester`, and `year` exist in multiple tables, and `NATURAL JOIN` may not handle them correctly.
4. **Better Alternative** – Use explicit `JOIN ... ON` conditions to ensure accurate joins.

Here's the corrected SQL query using explicit joins:

```
SELECT i.name, c.title

FROM instructor i

JOIN teaches t ON i.ID = t.ID

JOIN section s ON t.course_id = s.course_id

        AND t.sec_id = s.sec_id

        AND t.semester = s.semester

        AND t.year = s.year

JOIN course c ON s.course_id = c.course_id

WHERE s.semester = 'Spring' AND s.year = 2017;
```

4.2
Write the following queries in SQL:

a. Display a list of all instructors, showing each instructor's ID and the num ber of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.

Ans:

```
SELECT i.ID, COUNT(t.course_id) AS num_sections
FROM instructor i
LEFT JOIN teaches t ON i.ID = t.ID
GROUP BY i.ID;
```

b. Write the same query as in part a, but using a scalar subquery and not using outer join.

Ans:

```
SELECT i.ID,
      (SELECT COUNT(*)
       FROM teaches t
       WHERE t.ID = i.ID) AS num_sections
FROM instructor i;
```

c. Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to "—".

Ans:

```
SELECT
   s.course_id,
   s.sec_id,
   s.semester,
   s.year,
   COALESCE(i.ID, '—') AS instructor_id,
   COALESCE(i.name, '—') AS instructor_name
FROM
   section s
LEFT JOIN teaches t
   ON s.course_id = t.course_id
   AND s.sec_id = t.sec_id
   AND s.semester = t.semester
   AND s.year = t.year
LEFT JOIN instructor i
   ON t.ID = i.ID
```

d. Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show depart ments that have no instructors, and list those departments with an instruc tor count of zero.

Ans:

```
SELECT

   d.dept_name,

   COUNT(i.ID) AS instructor_count

FROM

   department d

LEFT JOIN instructor i

   ON d.dept_name = i.dept_name

GROUP BY

   d.dept_name;
```

4.3
Outer join expressions can be computed in SQL without using the SQL **outer join** operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the **outer join** expression.
a.
select * **from** *student* **natural left outer join** *takes*

Ans:
```
SELECT student.* , takes.*
FROM student
LEFT JOIN takes
ON student.ID = takes.ID;
```

b.
select * **from** *student* **natural full outer join** *takes*

Ans:

```
SELECT student.*, takes.*

FROM student
```

```
LEFT JOIN takes

ON student.ID = takes.ID


UNION


SELECT student.*, takes.*

FROM student

RIGHT JOIN takes

ON student.ID = takes.ID;
```

4.4
Suppose we have three relations $r(A, B)$, $s(B, C)$, and $t(B, D)$, with all attributes declared as **not null**.

a. Give instances of relations r, s, and t such that in the result of
$($*r* **natural left outer join** *s*$)$ **natural left outer join** *t*
attribute C has a null value but attribute D has a non-null value.

Ans:

Finding instances where attribute C is NULL but attribute D is non-null

We need to evaluate:

(r NATURAL LEFT OUTER JOIN s) NATURAL LEFT OUTER JOIN t(r \; \text{NATURAL LEFT OUTER JOIN} \; s) \; \text{NATURAL LEFT OUTER JOIN} \; t(rNATURAL LEFT OUTER JOINs)NATURAL LEFT OUTER JOINt

This means:

1. First, perform **rrr LEFT JOIN sss** → Some tuples from rrr may not find a match in sss, leading to NULL values in **C**.
2. Then, LEFT JOIN with ttt → Ensures all records from the first join are retained, and matches with ttt bring in attribute **D**.

Example Relations:

R (A, B)
1, X
2, Y
S(B, C)
Y, 100
t(B, D)
X, 200
Y, 300

Step-by-step Join Process:

1. **r NATURAL LEFT OUTER JOIN s**:

A	**B**	**C**
1	X	NULL
2	Y	100

   - **B = X** has no match in **s**, so **C = NULL**.
2. **Result LEFT JOIN ttt**:

A	**B**	**C**	**D**
1	X	NULL	200
2	Y	100	300

- We got a row **(1, X, NULL, 200)**, which satisfies our requirement: **C is NULL, but D is non-null**.

b. Are there instances of *r*, *s*, and *t* such that the result of
r **natural left outer join** (*s* **natural left outer join** *t*)

Ans:

Are there instances where:

r NATURAL LEFT OUTER JOIN (s NATURAL LEFT OUTER JOIN t)r \; \text{NATURAL LEFT OUTER JOIN} \; (s \; \text{NATURAL LEFT OUTER JOIN} \; t)rNATURAL LEFT OUTER JOIN(sNATURAL LEFT OUTER JOINt)

differs from

(r NATURAL LEFT OUTER JOIN s) NATURAL LEFT OUTER JOIN t?(r \;
\text{NATURAL LEFT OUTER JOIN} \; s) \; \text{NATURAL LEFT OUTER JOIN} \;
t?(rNATURAL LEFT OUTER JOINs)NATURAL LEFT OUTER JOINt?

Yes, there can be differences due to how NULL values propagate.

Consider this case:

r(A, B)
1, X

s(B, C)
(empty)

t(B, D)
X, 500

1. **sss NATURAL LEFT JOIN ttt**:
   - Since **s is empty**, this results in an **empty** table with only the schema **(B, C, D)**.
2. **rrr NATURAL LEFT JOIN (s NATURAL LEFT JOIN t)**:
   - Since the inner join produced nothing, the outer join with rrr keeps **r's tuples** but fills **C and D with NULLs**.

A	**B**	**C**	**D**
1	X	NULL	NULL

However, if we did **(r NATURAL LEFT JOIN s) NATURAL LEFT JOIN t**, we would have:

1. rrr LEFT JOIN sss → Keeps **all rrr tuples**, with **C = NULL** if no match.
2. Then LEFT JOIN with ttt → Matches XXX in ttt, filling **D**.

A	**B**	**C**	**D**
1	X	NULL	500

Since **NULL values in C propagate differently**, the results can be **different**!

4.5 Testing SQL queries: To test if a query specified in English has been correctly written in SQL, the SQL query is typically executed on multiple test databases, and a human check if the SQL query result on each test database matches the intention of the specification in English.

a. In Section 4.1.1 we saw an example of an erroneous SQL query which was intended to find which courses had been taught by each instructor; the query computed the natural join of *instructor*, *teaches*, and *course*, and as a result it unintentionally equated the *dept name* attribute of *instructor* and *course*. Give an example of a dataset that would help catch this particular error.

Ans:

```
SELECT i.name, t.course_id, c.title
FROM instructor i
JOIN teaches t ON i.ID = t.ID
JOIN course c ON t.course_id = c.course_id;
```

b. When creating test databases, it is important to create tuples in referenced relations that do not have any matching tuple in the referencing relation for each foreign key. Explain why, using an example query on the univer sity database.

Ans:

```
SELECT s.name, t.course_id
FROM student s
LEFT JOIN takes t ON s.ID = t.ID;
```

c. When creating test databases, it is important to create tuples with null values for foreign-key attributes, provided the attribute is nullable (SQL allows foreign-key attributes to take on null values, as long as they are not part of the primary key and have not been declared as **not null**). Explain why, using an example query on the university database.

Ans:

```
SELECT s.name, i.name AS advisor
FROM student s
LEFT JOIN instructor i ON s.ID = i.ID;
```

4.6
Show how to define the view *student grades* (*ID, GPA*) giving the grade-point average of each student, based on the query in Exercise 3.2; recall that we used a relation *grade points* (*grade*, *points*) to get the numeric points associated with a letter grades. Make sure your view definition correctly handles the case of *null* values for the *grade* attribute of the *take's* relation.

Ans:

```
CREATE VIEW student_gpa AS
SELECT
   s.ID AS student_id,
   COALESCE(
      SUM(gp.points * c.credits) / NULLIF(SUM(c.credits), 0),
   0) AS GPA
FROM
   student s
LEFT JOIN
   takes t ON s.ID = t.ID AND t.grade IS NOT NULL
LEFT JOIN
   course c ON t.course_id = c.course_id
LEFT JOIN
   grade_points gp ON t.grade = gp.grade
GROUP BY
   s.ID;
```

4.7
Consider the employee database of Figure 4.12. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

Ans:

Based on Figure 4.12 (assuming a standard employee database schema), here's the SQL DDL definition with referential integrity constraints:

```
CREATE TABLE employee (
  employee_id VARCHAR(10) PRIMARY KEY,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  street VARCHAR(100),
  city VARCHAR(50),
  state VARCHAR(2),
  zip VARCHAR(10),
  salary DECIMAL(10,2) CHECK (salary > 0),
  supervisor_id VARCHAR(10),
  department_number VARCHAR(10),
  FOREIGN KEY (supervisor_id) REFERENCES employee(employee_id)
    ON DELETE SET NULL,
  FOREIGN KEY (department_number) REFERENCES department(department_number)
    ON DELETE SET NULL
);

CREATE TABLE department (
  department_number VARCHAR(10) PRIMARY KEY,
  department_name VARCHAR(50) NOT NULL,
  manager_id VARCHAR(10),
  manager_start_date DATE,
  FOREIGN KEY (manager_id) REFERENCES employee(employee_id)
    ON DELETE SET NULL
);
```

```
CREATE TABLE project (
   project_number VARCHAR(10) PRIMARY KEY,
   project_name VARCHAR(100) NOT NULL,
   project_location VARCHAR(100),
   controlling_department VARCHAR(10),
   FOREIGN KEY (controlling_department) REFERENCES department(department_number)
      ON DELETE SET NULL
);

CREATE TABLE works_on (
   employee_id VARCHAR(10),
   project_number VARCHAR(10),
   hours DECIMAL(5,2) DEFAULT 0.00 CHECK (hours >= 0),
   PRIMARY KEY (employee_id, project_number),
   FOREIGN KEY (employee_id) REFERENCES employee(employee_id)
      ON DELETE CASCADE,
   FOREIGN KEY (project_number) REFERENCES project(project_number)
      ON DELETE CASCADE
);

CREATE TABLE dependent (
   employee_id VARCHAR(10),
   dependent_name VARCHAR(100),
   gender CHAR(1),
   relationship VARCHAR(20),
   birth_date DATE,
   PRIMARY KEY (employee_id, dependent_name),
   FOREIGN KEY (employee_id) REFERENCES employee(employee_id)
      ON DELETE CASCADE
);
```

4.8
As discussed in Section 4.4.8, we expect the constraint “an instructor cannot teach sections in two different classrooms in a semester in the same time slot” to hold.

a. Write an SQL query that returns all (*instructor*, *section*) combinations that violate this constraint.

Ans:
```
CREATE ASSERTION instructor_single_classroom_per_timeslot
CHECK (
  NOT EXISTS (
    SELECT 1
    FROM teaches t1
    JOIN teaches t2 ON t1.ID = t2.ID
            AND t1.semester = t2.semester
            AND t1.year = t2.year
            AND t1.time_slot_id = t2.time_slot_id
            AND (t1.course_id != t2.course_id OR t1.sec_id != t2.sec_id)
    JOIN section s1 ON t1.course_id = s1.course_id
            AND t1.sec_id = s1.sec_id
            AND t1.semester = s1.semester
            AND t1.year = s1.year
    JOIN section s2 ON t2.course_id = s2.course_id
            AND t2.sec_id = s2.sec_id
            AND t2.semester = s2.semester
            AND t2.year = s2.year
    WHERE (s1.building != s2.building OR s1.room_number != s2.room_number)
  )
);
```

b. Write an SQL assertion to enforce this constraint (as discussed in Sec tion 4.4.8, current generation database systems do not support such as sertions, although they are part of the SQL standard).

Ans:

```
CREATE ASSERTION instructor_single_classroom_per_timeslot

CHECK (

  NOT EXISTS (

    SELECT 1

    FROM teaches t1

    JOIN teaches t2 ON t1.ID = t2.ID

            AND t1.semester = t2.semester
```

```
                    AND t1.year = t2.year
                    AND t1.time_slot_id = t2.time_slot_id
                    AND (t1.course_id != t2.course_id OR t1.sec_id != t2.sec_id)
        JOIN section s1 ON t1.course_id = s1.course_id
                    AND t1.sec_id = s1.sec_id
                    AND t1.semester = s1.semester
                    AND t1.year = s1.year
        JOIN section s2 ON t2.course_id = s2.course_id
                    AND t2.sec_id = s2.sec_id
                    AND t2.semester = s2.semester
                    AND t2.year = s2.year
        WHERE (s1.building != s2.building OR s1.room_number != s2.room_number)
    )
);
```

4.9
SQL allows a foreign-key dependency to refer to the same relation, as in the following example:
create table *manager*
(*employee ID*
char(20),
manager ID
char(20),
primary key *employee ID*,
foreign key (*manager ID*) **references** *manager*(*employee ID*)
on delete cascade)
Here, *employee ID* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.

Ans:

What Happens When a Tuple is Deleted

1. **Direct Deletion**:

- The specified employee record is first deleted from the table

2. **Cascade Effect**:
    - The database then automatically searches for all records where the deleted employee's ID appears as a manager_ID (i.e., all employees who had the deleted employee as their manager)
    - Each of these "subordinate" records is also deleted
3. **Recursive Cascade**:
    - For each subordinate employee that gets deleted, the process repeats:
        - Find employees who had these now-deleted subordinates as their managers
        - Delete those employees as well
    - This continues recursively down the entire management hierarchy

4.10
Given the relations *a*(*name, address, title*) and *b*(*name, address, salary*), show how to express *a* **natural full outer join** *b* using the **full outer-join** operation with an **on** condition rather than using the **natural join** syntax. This can be done using the **coalesce** operation. Make sure that the result relation does not contain two copies of the attributes *name* and *address* and that the solution is correct even if some tuples in *a* and *b* have null values for attributes *name* or *address*.

Ans:

```
(
  SELECT a.name, a.address, a.title, b.salary
  FROM a LEFT JOIN b USING (name, address)
UNION
(
  SELECT b.name, b.address, a.title, b.salary
  FROM a RIGHT JOIN b USING (name, address)
LIMIT 0, 25;
```

4.11
Operating systems usually offer only two types of authorization control for data files: read access and write access. Why do database systems offer so many kinds of authorization?

Ans:

Database systems offer granular authorization controls because:

1. **Data Sensitivity**: Databases store more valuable/confidential data than files
2. **Operations Complexity**: Beyond read/write, databases support:
   - SELECT (read specific columns)
   - INSERT/UPDATE/DELETE (different write operations)
   - REFERENCES (foreign key constraints)
   - EXECUTE (stored procedures)
   - USAGE (schemas/sequences)
3. **Row-Level Security**: Need to restrict access to specific rows
4. **Column-Level Security**: Need to restrict access to specific columns
5. **View-Based Security**: Grant access through views rather than base tables
6. **Role Management**: Complex user roles require fine-grained permissions

4.12

Suppose a user wants to grant **select** access on a relation to another user. Why should the user include (or not include) the clause **granted by current role** in the **grant** statement?

Ans:

Including GRANTED BY CURRENT ROLE or WITH GRANT OPTION:

Reasons to include it:

- Allows the grantee to further grant privileges to others
- Useful for delegating administrative tasks
- Appropriate when you trust the grantee to manage permissions

Reasons not to include it:

- Security risk (privileges could be granted to unauthorized users)
- Makes permission tracking more difficult
- Against principle of least privilege

4.13

Consider a view *v* whose definition references only relation *r*.

- If a user is granted **select** authorization on *v*, does that user need to have **select** authorization on *r* as well? Why or why not?
- If a user is granted **update** authorization on *v*, does that user need to have **update** authorization on *r* as well? Why or why not?

Ans:

For SELECT on view v:

- No, the user doesn't need SELECT on r
- The view owner must have SELECT on r

- The DBMS checks the view owner's privileges, not the user's

For UPDATE on view v:

- Yes, the user typically needs UPDATE on r
- View updates translate to base table updates
- Some DBMS may allow it if view is updatable and user has privileges through view owner
- Depends on the DBMS implementation and view definition

The key difference is that SELECT through a view acts as a filter using the view owner's privileges, while UPDATE must modify the underlying data which requires proper authorization.

4.14
Consider the query
select *course id*, *semester*, *year*, *sec id*, **avg** (*tot cred*)
from *takes* **natural join** *student*
where *year* = 2017
group by *course id*, *semester*, *year*, *sec id*
having count (*ID*) >= 2;
Explain why appending **natural join** *section* in the **from** clause would not change
the result.

Ans:

The original query:

SELECT course_id, semester, year, sec_id, avg(tot_cred)

FROM takes NATURAL JOIN student

WHERE year = 2017

GROUP BY course_id, semester, year, sec_id

HAVING count(ID) >= 2;

The reason behind the appending **natural join** *section* in the **from** clause would not change the result.

1. **Grouping Columns Already Determine Section Identity**:
    - The query already groups by course_id, semester, year, and sec_id
    - These attributes uniquely identify a section in the database schema

2. **No Additional Filtering**:
    - Joining with the section table doesn't add any new WHERE conditions
    - It doesn't exclude any rows that would affect the grouping or averages
3. **No Additional Aggregated Data**:
    - The avg(tot_cred) is calculated from student data already present
    - The section table doesn't contain any attributes that would affect this average
4. **NATURAL JOIN Behavior**:
    - The join would only add attributes from section that aren't already in the result
    - But since we're not selecting any additional attributes, this has no effect
5. **Count Condition Remains Unchanged**:
    - The HAVING count(ID) >= 2 counts student IDs in each group
    - Joining with section doesn't change the number of students per section

4.15
Rewrite the query
select *
from *section* **natural join** *classroom*
without using a natural join but instead using an inner join with a **using** condition.

Ans:

Can be rewritten using INNER JOIN with USING as follows:

```
SELECT *

FROM section INNER JOIN classroom

USING (building, room_number);
```

4.16

Write an SQL query using the university schema to find the ID of each student who has never taken a course at the university. Do this using no subqueries and no set operations (use an outer join).

Ans:

```
SELECT s.ID

FROM student s

LEFT JOIN takes t ON s.ID = t.ID

WHERE t.ID IS NULL;
```

4.17
Express the following query in SQL using no subqueries and no set operations.
select *ID*
from *student*
except
select *s id*
from *advisor*
where *i ID* **is not null**

Ans:

To find student IDs who don't have an advisor (without using subqueries or set operations), we can use a LEFT JOIN approach:

```
SELECT s.ID

FROM student s

LEFT JOIN advisor a ON s.ID = a.s_ID

WHERE a.s_ID IS NULL;
```

Explanation:

1. **LEFT JOIN** connects all students with their advisor relationships
2. **WHERE a.s_ID IS NULL** filters for students with no advisor record
3. This is equivalent to the EXCEPT operation in the original query but:
   - Uses only joins
   - No subqueries
   - No set operations

4.18
For the database of Figure 4.12, write a query to find the ID of each employee with no manager. Note that an employee may simply have no manager listed or may have a *null* manager. Write you

Ans:

```
SELECT employee_ID

FROM employee

WHERE manager_ID IS NULL;
```

Explanation:

1. This query selects all employee IDs where:
   - The manager_ID field is explicitly NULL (no manager assigned)
   - OR the employee has no manager record at all (implied by NULL)
2. The query handles both cases of "no manager":
   - Employees with NULL in the manager_ID column
   - Employees missing from the manager table entirely (covered by the NULL check)
3. Alternative version that's more explicit about both conditions:

4.19
Under what circumstances would the query

select *
from *student* **natural full outer join** *takes*
natural full outer join *course*
include tuples with null values for the *title* attribute?

Ans:

```
SELECT *

FROM student NATURAL FULL OUTER JOIN takes

NATURAL FULL OUTER JOIN course
```

Cases Where title Would Be NULL:

1. **Students Who Haven't Taken Any Courses**
   - When a student exists in the student table but has no matching records in takes
   - The join with course would produce NULLs for all course attributes (including title)
   - Example: A newly enrolled student who hasn't registered for classes yet
2. **Courses That Exist But Have No Students Enrolled**
   - When a course exists in the course table but has no matching records in takes
   - The student attributes would be NULL, but the course title would be present
   - *(Wait, this would actually show the title but NULL student info - opposite of what you asked)*
3. **Mismatched Joins Between takes and course**
   - If a student took a course that no longer exists in the course table (referential integrity violation)
   - The course attributes (including title) would be NULL while student and takes info would exist

4.20
Show how to define a view *tot credits* (*year, num credits*), giving the total number of credits taken in each year.

Ans:
```
CREATE VIEW tot_credits(year, num_credits) AS
SELECT year, SUM(credits)
FROM takes NATURAL JOIN course
GROUP BY year;
```

4.21
For the view of Exercise 4.18, explain why the database system would not allow a tuple to be inserted into the database through this view.

Ans:
The view tot_credits is not updatable because:

1. It contains aggregation (SUM)
2. It uses GROUP BY
3. It joins multiple tables
4. The view doesn't preserve the base table's keys
   These characteristics make it impossible for the DBMS to determine how to propagate inserts to the underlying tables.

4.22
Show how to express the **coalesce** function using the **case** construct.

Ans:
```
SELECT
   CASE
      WHEN column1 IS NOT NULL THEN column1
      WHEN column2 IS NOT NULL THEN column2
      ELSE column3   -- or any default value like NULL, 'default', etc.
   END AS result
FROM your_table_name;
```

4.23
Explain why, when a manager, say Satoshi, grants an authorization, the grant should be done by the manager role, rather than by the user Satoshi.

Ans:
Granting by role rather than individual user:

1. Maintains continuity when personnel change
2. Follows principle of least privilege
3. Makes auditing and accountability clearer

4. Prevents accidental privilege retention when users leave
5. Aligns with organizational policies rather than personal authority

4.24
Suppose user *A*, who has all authorization privileges on a relation *r*, grants**select** on relation *r* to **public** with grant option. Suppose user *B* then grants **select** on *r* to *A*. Does this cause a cycle in the authorization graph? Explain why.

Ans:
No, this doesn't create a cycle because:

1. The grant from A to public is a one-way relationship
2. B's grant back to A is redundant (A already has privileges)
3. The authorization graph tracks privilege flows, not reciprocal grants
4. Public is a special group, not a node that can receive privileges

4.25
Suppose a user creates a new relation *r*1 with a foreign key referencing another relation *r*2. What authorization privilege does the user need on *r*2? Why should this not simply be allowed without any such authorization?

Ans:
The user needs REFERENCES privilege on r2 because:

1. It ensures the user can verify referential integrity
2. Prevents unauthorized creation of constraints
3. Maintains data consistency across relations
4. Gives r2's owner control over who can reference their data
   This shouldn't be allowed without authorization as it could enable:

- Creation of invalid constraints
- Circumvention of access controls
- Potential denial-of-service through constraint validation

4.26
Explain the difference between integrity constraints and authorization con straints.

Ans:

Integrity Constraints:

- Ensure data correctness (e.g., NOT NULL, FOREIGN KEY)
- Apply to all users equally
- Enforce business rules and data relationships
- Examples: Primary keys, domain constraints

Authorization Constraints:

- Control data access (e.g., GRANT, REVOKE)
- User-specific permissions
- Govern who can perform what operations
- Examples: SELECT privileges, UPDATE restrictions

Key difference: Integrity protects data quality; authorization protects data security.