Unit 3 Syllabus

Constraints, Views and SQL

Constraints,

Types of constraints,

Integrity constraints,

Views: Introduction to views,

Data independence,

Security, updates on views,

Comparison between tables and views

- *SQL:* data definition,
- Aggregate function,
- Null values,
- Nested sub queries,
- Joined relations
- Triggers.

Unit 3

Chapter 1: Constraints, Views and SQL

- Constraints, types of constraints,
- Integrity constraints,

- *Views:* Introduction to views, data independence, Security, updates on views

- SQL: data definition, aggregate function, Null values, Nested sub queries, Joined relations, Triggers.

- **Constraints**
- Constraints are limitations that are been enforced on particular column of an relation(table).
- **Types of constraints:**
  - Domain Constraints
  - Primary Key
  - Unique
  - Not null
  - Check
  - Default
  - Foreign key (Referential Integrity Constraints)

- *Domain Constraints*: It specifies each value for an attribute must be an atomic value from the domain.
- Example: Character not allowed in Age column

- *Primary Key Constraints*: It is used to Uniquely identify a row as well as not null value in a relational table.

✔ Only one primary key allowed for one table.

✔ Checks for unique and not null constraints by default.

- Example: create table dept(dno int constraint P primary key, dname char(10));

- *Unique constraint*: Checks that value entered in the column is unique.
- Example: create table student(roll_no number(3) constraint U unique, name char(10));
- *Not null constraint*: Checks that the column must always have some value.
- Example: create table student(roll_no number(3), name char(10) constraint N not null);
- *Default constraint*: Provides a default value for a particular column.

- *Check Constraint*: It enforces a restriction on the value entered by the user.
- Examples:
- 1. create table emp(id number(4), sal number(7) constraint C check(sal between 1000 and 5000000)), name char(10));
- 2. create table stud(roll int, name char(10) constraint name_chk check(name=upper(name)));
- 3. create table employee(eno number(3), dno number(4) constraint CHK check(dno in (10,20,30)));

- *Foreign Key Constraint (Referential Integrity Constraint)*: It can reference primary or unique key.

✔ There can be more than one foreign key for one table.

✔ Datatype of referencing and referred column should be same.

✔ The parent record cannot be deleted when child record exists.

To delete parent and child record both, use ‘on delete cascade’.

- **Example:** Main table(parent table):-
- create table dept(dno number(3) primary key, dname char(10));
- References(Child table):-
- create table emp(eno number(5), deptno number(3) constraint F references dept(dno) on delete cascade, ename char(10));

- To add:
- alter table tablename add constraint C *constraint*
- *Constraints can be added in two ways:-*
- Column level: specified for particular columns
- Table level: specified at last after all the columns
- Example-create table A1(roll int, name char(10), constraint U unique(roll));

- To drop
- alter table tablename drop constraint C;

- **Integrity Constraints**

- Integrity constraints ensure that changes made to the database by authorized uses do not result in a loss of data consistency.
- Thus, integrity constraints guard against accidental damage to the database.

- Examples of integrity constraints are:
- No two instructors can have the same instructor ID.
- every department name in the *course* relation must have a matching department name in the *department* relation.
- The budget of a department must be greater than $0.00.

- In general, an integrity constraint can be an arbitrary predicate pertaining to the database.
- However, arbitrary predicates may be costly to test. Thus, most database systems allow one to specify integrity constraints that can be tested with minimal overhead.

- Data Integrity: validates data before storing in columns of table. (Unique)
- Entity Integrity: ensure each row is uniquely identifiable entity. (Primary key, Unique)
- Domain Integrity: ensure data values should follow defined rules, range, format. (Check, Unique, Default)
- Referential Integrity: ensures the relationships between tables remain preserved as data is inserted, deleted and modified. (Foreign key, Check)
- User Defined Integrity: allows to define specific business rules that do not fall in other categories. (Create table, procedures, triggers)

Views

- SQL allows a “virtual relation” to be defined by a query, and the relation conceptually contains the result of the query.
- The virtual relation is not pre computed and stored, but instead is computed by executing the query whenever the virtual relation is used.
- i.e. A view can be thought of as a virtual table that consists of only specified rows and columns from table or tables based on condition specified at the time of its creation. It allows customization of perception of each user.
- Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **View**.

Benefits of using Views:

- ✔ Security
- ✔ Hides Data Complexity
- ✔ Present data in different perspective
- ✔ Isolate applications from making changes in definition of base tables.

- Generally view are created to use in custom applications, report generation, ad hoc queries, etc. as these most of the times need only a VIEW of database!

View Definition

- The general syntax for creation is:

  create view *v* **as** <query expression>;

- i.e. CREATE VIEW view_name AS

  SELECT column1, column2,..

  FROM table_name

  WHERE condition;

- To *describe and display view*
- Describe view_name;
- Select * from view_name;

- To *insert into* view

  insert into view_name (column_name1,

  column_name2,…) values (value1, value2,…);

- To *delete* view

  delete view view_name;

- **Data Independence**

Security

- Views help to enhance the security feature in the database system by limiting the data presented to the end user.
- A view uses the results of a database query to dynamically populate the contents of an artificial database table.
- Thus a view can be used to give the access of non sensitive data to a specific groups of users.
- We can create Read Only Views, grant access to only these views to end users, which help restricting Insert, Update and Delete actions.

Updates on Views

- Different database systems specify different conditions under which they permit updates on view relations.
- An SQL view is said to be **updatable,** if the following conditions are all satisfied by the query defining the view:

✔ The **from** clause has only one database relation.

✔ The **select** clause contains only attribute names of the relation and does not have any expressions, aggregates or **distinct** specification.

✔ Any attribute not listed in **select** clause can be set to *null*; i.e., it does not have a **not null** constraint & is not part of a primary key.

✔ The query does not have a **group by** or **having** clause.

- Under these constraints, the **update, insert,** and **delete** operations would be allowed on the following view:

  create view *history_instructors* **as**

  select *

  from *instructor*

  where *dept name*= ‘History’;

- Syntax for updating view

  update view_name set column_name=‘value’ where (condition);

  OR

  CREATE OR REPLACE VIEW view_name AS

  SELECT….

Comparison between Tables and Views

Tables	View
Table is a primary storage for storing data in relational database management system.	A view is a virtual table whose contents are defined by a query.
When data is huge, there may be complexity in tables.	A view hides the complexity of the database tables from end users.
Space required by tables in memory is more.	Space required by views in memory very less.
Tables can be created independently.	Views are always created on tables.
Table is designed as limited number of column and unlimited number of rows.	View is designed as virtual table that is extracted from a database.
Multiple tables are required to store linked data and records.	**View can integrate several tables in to one virtual table.**

SQL

- IBM developed the original version of SQL, originally called Sequel in early 1970s.
- It evolved and its name changed to SQL (Structured Query Language).
- The SQL language has several parts:

- DDL
- DML
- Integrity
- View Definition
- Transaction Control
- Authorization

Data Definition

- The set of relations in a database must be specified to the system by means of a Data Definition Language(DDL).

- **Basic Types**
- The SQL standard supports a variety of built-in domain types:

- char

- varchar2

- int

- number

- float

Basic Schema Definition

- We define an SQL relation by using the **create table** command.
- A newly created relation is empty initially. We can use the **insert** command to load data into relation.
- The values are specified in the *order* in which the corresponding attributes are listed in the relation schema.
- We can use the **delete** command to delete tuples from a relation. Other forms of the delete command allow specific tuples to be deleted.

- To remove a relation from a SQL database, we use **drop table** command. The **drop table** command deletes all information about the dropped relation from the database.
- We use the **alter table** command to add attributes to an existing relation.

- *The Rename Operation*
- SQL provides a way of renaming the attributes of a result relation.
- It uses the **as** clause, taking the form:
- *old-name* **as** *new-name*
- The **as** clause can appear in both the **select** and **from** clauses.

- Example (AS clause for Select)
- If we want the attribute name *name* to be replaced with the name *instructor name,* we can rewrite the preceding query as:

  select name **as** *instructor_name, course_id*

  from *instructor, teaches*

  where *instructor.ID = teaches.ID;*
- Example (AS clause for From)
- "For all instructor in university who have taught some course, find their names & course ID of all courses they taught."

  select *T.name, S.course.id*

  from *instructor* **as** *T*, *teaches* **as** *S*

  where *T.ID = S.ID*;

- Example: Renaming for Self Cartesian Product
- “Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.”
- **select distinct** *T.name*
- **from** *instructor* **as** *T*, *instructor* **as** *S*
- **where** *T.salary > S.salary* **and** *S.dept.name = ‘*Biology*’;*

- An identifier, such as *T* and *S*, that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but is also commonly referred to as a **table alias,** or a **correlation variable**, or a **tuple variable.**

Basic Structure of SQL Queries

- The basic structure of an SQL query consists of three clauses: *select, from and where.*
- The relations listed in the **from** clause are taken as its input, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result.

- A typical SQL query has the form:

  select *A1, A2, . . . , An*

  from *r1,r2*, . . . ,*rm*

  where *P*

Where Clause Predicates

- SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.
- If we wish to find the names of instructors with salary amounts between $90,000 and $100,000, we use the **between:**
- **select** *name* **from** *instructor*
- **where** *salary* **between** 90000 **and** 100000;
- Instead of:
- **select** *name* **from** *instructor*
- **where** *salary* <= 100000 and *salary* >=90000;
- Similarly, we can use the **not between** comparison operator.

Null Values

- Null (or NULL) is a special marker used in Structured Query Language (SQL) to indicate that a data value does not exist in the database.
- Introduced by the creator of the relational database model, E. F. Codd, SQL Null serves to fulfill the requirement that all true relational database management systems (RDBMS) support a representation of *“missing information and inapplicable information”*.
- In SQL, NULL is a reserved word used to identify this marker.

- Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.
- The result of an arithmetic expression (involving, for example +, -, *, or /) is null if any of the input values is null.
- Comparisons involving nulls are more of a problem.
- Since the predicate in a **where** clause can involve Boolean operations such as **and, or** and **not** on the results of comparisons, the definitions of Boolean operations are extended to deal with the value **unknown.**

- **and:** The result of
  - ✔ *true* **and** *unknown* is *unknown*,
  - ✔ *false* **and** *unknown* is *false*,
  - ✔ *unknown* **and** *unknown* is *unknown*.
- **or:** The result of
  - ✔ *true* **or** *unknown* is *true*,
  - ✔ *false* **or** *unknown* is *unknown*,
  - ✔ while *unknown* **or** *unknown* is *unknown*.
- **not:** The result of
  - ✔ **not** *unknown* is *unknown*

- Following operations can be performed with NULL:

1. Inserting NULL value into a table.
2. Updating records having NULL values.
3. Deleting records having NULL values.
4. Displaying records having NULL values.

 Inserting NULL values into a table.

- insert into emp values(1,null,’Mumbai’,null);
- insert into emp values(2,’Alice’,’Banglore’,’’);
- insert into emp (eid,ename) values(3,’Bob’);

 Updating records having NULL values

- update emp set salary=25000 where ename IS NOT NULL;
- update emp set salary=00 where ename IS NULL;

 Deleting records having NULL values.

- delete from emp where ename IS NULL;
- delete from emp where ename IS NOT NULL;

 Displaying records having NULL values.

- select * from emp where ename IS NOT NULL;
- select * from emp where ename IS NULL;

Aggregate Functions

- *Aggregate functions* are functions that take a collection (a set or multiset) of values as input and return a single value.
- SQL offers five built-in aggregate functions:

1. Average: **avg()**
2. Minimum: **min()**
3. Maximum: **max()**
4. Total: **sum()**
5. Count: **count()**

Basic Aggregation Examples

- Find the average salary of instructors in the Computer Science department.

  select avg (*salary*) **as** *avg_salary* **from** *instructor*

  where *dept_name* = ‘Comp. Sci.’;

- Find the total number of instructors who teach a course in the Spring 2010 semester.

  select count (**distinct** *ID*) **from** *teaches*

  where *semester* = ‘Spring’ **and** *year* = 2010;

- Find the number of tuples in the *course* relation

  select count (*) **from** *course*;

Aggregation with Grouping

- To apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify **group by** clause.
- The attribute or attributes given in the **group by** clause are used to form groups.
- Tuples with the same value on all attributes in the **group by** clause are placed in one group.

Tuples of the *instructor* relation, grouped by the *dept_name* attribute.

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

- Example
- Find the average salary in each department.

  select *dept_name,* **avg**(*salary*) **as** *avg_salary* **from** *instructor*

  group by *dept_name;*

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

The Having Clause

- To state a condition that applies to groups rather than to tuples.
- For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause.
- To express such a query, we use the **having** clause of SQL.
- Q. Find the average salary of instructors in those departments where the average salary is more than $42,000.

- Query

  select *dept name*, **avg** (*salary*) **as** *avg salary* **from** *instructor*
  group by *dept name*
  having avg (*salary*) > 42000;

dept_name	*avg(avg_salary)*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

The result relation for the query "Find the average salary of instructors in those departments where the average salary is more than $42,000."

- The following sequence of operations:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.

2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.

3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.

4. The having clause, if it is present, is applied to each group; the groups that do not satisfy the having clause predicate are removed.

5. The select clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

Set Operations

- The SQL operations **union, intersect,** and **except** operator on relations and corresponds to the mathematical set-theory operations $\cup$, $\cap$ and $-$
- Example:
- The set of all courses taught in the Fall 2009 semester:

  select *course_id* **from** *section*

  where *semester* = *'*Fall' **and** *year*= 2009;

- The set of all courses taught in the Spring 2010 semester:

  select *course_id* **from** *section*

  where *semester* = 'Spring' **and** *year*= 2010;

- c1: courses taught in Fall 2009.

course_id
CS-101
CS-347
PHY-101

- c2: courses taught in Spring 2010

course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

The Union Operation

- Find the set of all courses taught either in Fall 2009 or in Spring 2010, or both,

  (**select** *course id*
  from *section*
  where *semester* = ’Fall’ **and** *year*= 2009)
  union
  (**select** *course id*
  from *section*
  where *semester* = ’Spring’ **and** *year*= 2010);

- The **union** operation automatically eliminates duplicates, to retain all duplicates, we must write **union all.**

The result relation for *c1* union *c2*.

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

The Intersect Operation

- To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:
  (**select** *course id*
  from *section*
  where *semester* = 'Fall' **and** *year*= 2009)
  intersect
  (**select** *course id*
  from *section*
  where *semester* = 'Spring' **and** *year*= 2010);
- The **intersect** operation automatically eliminates duplicates, to retain all duplicates, we must write **intersect all** in place of **intersect**

- **The Except Operation**
- To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:

  (**select** *course id*
  from *section*
  where *semester* = ’Fall’ **and** *year*= 2009)
  except
  (**select** *course id*
  from *section*
  where *semester* = ’Spring’ **and** *year*= 2010);
- The except operation outputs all tuples from its first input that do not occur in the second input; i.e., it performs set difference.

- The operation automatically eliminates duplicates in the inputs before performing set difference.
- If we want to retain duplicates, we must write **except all** in place of **except**

course_id
CS-347
PHY-101

The result relation for *c1* except *c2*.

- **Nested Subqueries**

- SQL has an ability to nest queries within one another.
- A subquery is a **select-fromwhere** expression that is nested within another query
- SQL executes innermost subquery first, then next level.
- A common use of subqueries is to perform set comparisons, etc, by nesting subqueries in the **where** clause.
- Nesting of subqueries can also be done in the **from** clause and also scalar subqueries can appear wherever an expression returning a value can occur.

Set Membership

- SQL allows testing tuples for membership in a relation.

- The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause.

- The **not in** connective tests for the absence of set membership.

Q. Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters.

✔ query using the **in** connective of SQL:

- Subquery:

  (**select** *course id* **from** *section*
  where *semester* = 'Spring' **and** *year*= 2010)

- The resulting query is

- **select distinct** *course id* **from** *section*
  where *semester* = 'Fall' **and** *year*= 2009
  and *course id* **in** (**select** *course id* **from** *section*
  where *semester* = 'Spring' **and** *year*= 2010);

- We use the **not in** construct in a way similar to the **in** construct.
- Example: To find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester:

  select distinct *course id* **from** *section*
  where *semester* = 'Fall' **and** *year*= 2009
  and *course id* **not in** (**select** *course id* **from** *section*
  where *semester* = 'Spring' **and** *year*= 2010);

- The **in** and **not in** operators can also be used on enumerated sets.
- Example: selects the names of instructors whose names are neither "Mozart" nor "Einstein".

  select distinct *name* **from** *instructor*
  where *name* **not in** ('Mozart', 'Einstein');

- **Set Comparison**
- Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.

  select *name* **from** *instructor*
  where *salary* > **all** (**select** *salary* **from** *instructor*
  where *dept name* = 'Biology');

50000	>some	60000
60000		70000
70000		
80000		

- SQL also allows <, >, <=, >=, =, and <> comparisons.

 some

 exists, not exists

 unique, not unique

- **Scalar Subqueries**

- SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute

- Such subqueries are called **scalar subqueries**.

- For example, a subquery can be used in the **select** clause

Q. Lists 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;
```

- Scalar subqueries can occur in **select**, **where**, and **having** clauses.
- Scalar subqueries may also be defined without aggregates.

Joined Relations

- In SQL, when two or more tables are involved in a query it is called as Joins.
- There are different types of Join queries which can be used to display the data from multiple tables.
- Example: Natural Join, Inner Join, Outer Join, etc.
- Join Conditions
- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where.**

- ‘Student’ Relation

  ID, Name, dept_name, tot_cred

- ‘Takes’ Relation

  course_id, sec_id, semester, year, grade, ID

select * **from** *student* **join** *takes*

on *student.ID = takes.ID*

- The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.
- Equivalent query:

  select * **from** *student, takes*

  where *student.ID = takes.ID;*

- If we want ID to be displayed once:

  select *student.ID* **as** *ID, name, dept_name, tot_cred,*

  course_id,sec_id,semester, year, grade

  from *student* **join** *takes* **on** *student.ID= takes.ID*;

Cartesian Product

- For every row of table 1 it will display all the rows of table 2.
- Each tuple in *instructor* is combined with *every* tuple in *teaches.*
- The **from** clause by itself defines a Cartesian product of the relations listed in the clause.
- The result relation has all attributes from all the relations in the **from** clause.
- Since the same attribute name may appear in both *ri* and *rj*, we prefix the name of the relation from which the attribute originally came, before the attribute name.

select *instructor.ID, name, course_id, sem, year*

from *instructor, teaches*;

Natural Join

- It considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations.
- i.e. The rows from both tables will be taken, for the column which is common, having matching values.

- Tuple in *instructor* is combined with tuple in *teaches* for the common attribute *ID* having same values.

select * from *instructor* **natural join** *teaches*;

Inner Join

- **Inner join** is equivalent to **natural join.**
- Also the *inner* keyword is optional.
- Thus,

  select * **from** *student* **join** *takes* **using** (*ID*);
- Is equivalent to:

  select * from *student* **inner join** *takes* **using** (*ID*);
- The **‘using’** keyword is used to specify the attribute which is to be taken for joining the different relations.

The Outer Join

- The **outer join** operation preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.
- Three forms of outer join:
- The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation.
- The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.
- The **full outer join** preserves tuples in both relations.

- Display a list of all students, displaying their *ID,* and *name, dept_name* and *tot_cred,* along with the courses that they have taken. The following is the SQL query-
- **select * from** *student* **natural join** *takes*;
- Problem:
- Suppose that there is some student who takes no courses. Then the tuple in the *student* relation for that particular student would not satisfy the condition of a natural join with any tuple in the *takes* relation, and that student's data would not appear in the result.

- Solution: Outer joins
- Left Outer Join

  select * from *student* **left outer join** *takes;*
- Right Outer Join
- **select * from** *takes* **right outer join** *student;*
- Full Outer Join

  select * from *student* **full outer join** *takes;*

Join types and join conditions.

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

Triggers

- A trigger is a PL/SQL block structure which is fired when a DML statement like Insert, Update and Delete is executed on a database table.
- A trigger is triggered automatically when the statement is executed.

 Components:

- Events- The trigger activities on events like Insert, Update, Delete
- Condition- If condition is specified & its true, trigger is fired.
- Action- The actual PL/SQL block to be executed is specified.

- Events
- Insert,
- Update and
- Delete

- Event time
- Before, After

- Granularity (Level)
- For each Row,
- For each Statement

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE/AFTER/INSTEAD OF} event
[OF col_name]
ON table_name
[FOR EACH ROW/FOR EACH STATEMENT]
[WHEN (condition)]
BEGIN
    statements;
END;
/
```

Example:

1. Create a statement level trigger that is fired after inserting rows in Emp table

- CREATE TRIGGER ct1
- AFTER INSERT ON Emp
- BEGIN
- DBMS_OUTPUT.PUT_LINE(‘The row is inserted in the table’);
- END;

2. Create a row level trigger which should fire before updating the rows of the Emp table and allow to update only if new value of salary is greater than 5000 else retain the old value.

```
CREATE TRIGGER ct2
BEFORE UPDATE OF Sal ON Emp
FOR EACH ROW
WHEN (new.sal<5000)
BEGIN
DBMS_OUTPUT.PUT_LINE('The row is not updated');
:new.sal := :old.sal;
END;
```