FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Introduction to SQL:
Schema definition, Constraints, Queries

DATABASES - lectures

On SQL

- **SQL** stands for Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language)
- SQL is the standard language for commercial relational DBMSs
  - SQL (ANSI 1986), called SQL-86 or SQL1.
  - Revised and much expanded standard called SQL2 (also called SQL-92)
  - SQL-99
- SQL is a comprehensive database language: It has statements for data definitions, queries, and updates.

SQL: DDL and DML

- SQL is
  - **DDL** (Data Definition Language) and
  - **DML** (Data Manipulation Language).
- In addition, it has facilities for:
  - Defining views on the database
  - Specifying security and authorization
  - Defining integrity constraints
  - Specifying transaction control
  - It also has rules for embedding SQL statements into a general-purpose programming language such as Java, Cobol, or C/C++.

Data Definition Language - DDL

- These statements are used to:
  - CREATE,
  - DROP,
  - or ALTER

  the descriptions of the tables (relations) in the database

Each database must be stored in the physical memory of the system in order to be used!

CREATE SCHEMA

- Specifies a new database schema by giving it a name
- An SQL schema is identified by a schema name, and includes
  - authorization identifier to indicate the user or account who owns the schema
  - descriptors for each element in the schema
    - Tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema
- A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions.
- Alternatively, the schema can be assigned a name and authorization identifier, and the elements can be defined later.
- For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'.

```
CREATE SCHEMA COMPANY AUTHORIZATION Jsmith;
```

CREATE TABLE

- Specifies a new base relation by giving it a name, and specifying each of its attributes and their data types (INTEGER, FLOAT, DECIMAL(i,j), CHAR(n), VARCHAR(n))
- Example statement for creating the table for the Department relation in the COMPANY example:

```
CREATE TABLE DEPARTMENT (
   DNAME       VARCHAR(10),
   DNUMBER          INTEGER,
   MGRSSN           CHAR(9),
   MGRSTARTDATE     CHAR(9)   );
```

CREATE TABLE

- The CREATE TABLE command can be used for specifying:
  - The primary key attributes (PRIMARY KEY)
  - Attribute uniqueness (UNIQUE)
  - If an attribute is compulsory (NOT NULL)
  - Referential integrity constraints – foreign keys (FOREIGN KEY + REFERENCES)

```
CREATE TABLE DEPT (
    DNAME         VARCHAR(10)        NOT NULL,
    DNUMBER       INTEGER            NOT NULL,
    MGRSSN        CHAR(9),
    MGRSTARTDATE  CHAR(9),
    PRIMARY KEY (DNUMBER),
    UNIQUE (DNAME),
    FOREIGN KEY (MGRSSN) REFERENCES EMP(SSN)   );
```

Referential integrity options

- We can specify RESTRICT, CASCADE, SET NULL or SET DEFAULT on referential integrity constraints (foreign keys)

```
CREATE TABLE DEPT (
   DNAME        VARCHAR(10)        NOT NULL,
   DNUMBER      INTEGER            NOT NULL,
   MGRSSN             CHAR(9),
   MGRSTARTDATE       CHAR(9),
   PRIMARY KEY (DNUMBER),
   UNIQUE (DNAME),
   FOREIGN KEY (MGRSSN) REFERENCES EMP(SSN)
      ON DELETE SET DEFAULT
      ON UPDATE CASCADE  );
```

Referential integrity options (2)

```
CREATE TABLE EMP(
    ENAME          VARCHAR(30) NOT NULL,
    ESSN           CHAR(9),
    BDATE          DATE,
    DNO            INTEGER,
    SUPERSSN       CHAR(9),
    PRIMARY KEY (ESSN),
    FOREIGN KEY (DNO) REFERENCES DEPT(DNO)
          ON DELETE SET DEFAULT
          ON UPDATE  CASCADE,
    FOREIGN KEY (SUPERSSN) REFERENCES EMP(SSN)
          ON DELETE SET NULL
          ON UPDATE CASCADE   );
```

Attribute constraints

- A constraint on an attribute or a group of attributes can be given a name using the keyword **CONSTRAINT**
- If an attribute has a default value it can be specified using the keyword **DEFAULT**
  - If no default is specified and the attribute is a NOT NULL attribute then the default value is 0
  - Otherwise the default value is NULL
- Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition
- *Table constraints can be specified through additional CHECK* clauses at the end of a CREATE TABLE statement. These can be called **tuple-based** constraints

Attribute constraints

```
CREATE TABLE EMP(
    ENAME        VARCHAR(30) NOT NULL,
    ESSN         CHAR(9),
    BDATE        DATE,
    DNO          INTEGER  DEFAULT 1
       CHECK (DNO > 0 AND DNO < 21),
    SUPERSSN     CHAR(9),
    CONSTRAINT EMPSSN
       PRIMARY KEY (ESSN),
    CONSTRAINT EMPDEPT
       FOREIGN KEY (DNO) REFERENCES DEPT(DNO)
          ON DELETE SET DEFAULT
          ON UPDATE  CASCADE,
    CONSTRAINT EMPSUPSSN
       FOREIGN KEY (SUPERSSN) REFERENCES EMP(SSN)
          ON DELETE SET NULL
          ON UPDATE CASCADE   );
```

Additional Data Types in SQL2 and SQL-99

Has DATE, TIME, and TIMESTAMP data types

- **DATE:**
  - Made up of year-month-day in the format yyyy-mm-dd
  - 2015-11-09
- **TIME:**
  - Made up of hour:minute:second in the format hh:mm:ss
  - 13:15:30
- **TIME(i):**
  - Made up of hour:minute:second plus i additional digits specifying fractions of a second
  - format is hh:mm:ss.iiiii
  - Ex. for i=6 -> 13:15:30.648320

Additional Data Types in SQL2 and SQL-99

- **TIMESTAMP:**
  - Has both DATE and TIME components with a blank space between
  - 2015-11-09 13:15:30
- **INTERVAL:**
  - Specifies a relative value rather than an absolute value
  - Can be DAY/TIME (dd hh:mm:ss) intervals or YEAR/MONTH (yyyy-mm) intervals
  - Can be positive or negative when added to or subtracted from an absolute value, the result is an absolute value

DROP TABLE

- Used to remove a relation (base table) and its definition
- The relation can no longer be used in queries, updates, or any other commands since its description no longer exists
- If a table is referenced in constraints, then the CASCADE option should be used (after the table name)
- Example:

```
DROP TABLE DEPENDENT CASCADE;
```

ALTER TABLE

- Used to add an attribute to one of the base relations
  - The new attribute will have NULLs in all the tuples of the relation right after the command is executed; hence, the NOT NULL constraint is not allowed for such an attribute
- Example:

```
ALTER TABLE EMPLOYEE ADD JOB VARCHAR(12);
```

- The database users must still enter a value for the new attribute JOB for each EMPLOYEE tuple.
  - This can be done using the UPDATE command.

Retrieval Queries in SQL

- SQL has one basic statement for retrieving information from a database; the **SELECT** statement
- Important distinction between SQL and the formal relational model:
  - SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values
  - Hence, an SQL relation (table) is a **multi-set** (sometimes called a **bag**) of tuples; it is *not* a set of tuples

SQL relations can be constrained to be sets by specifying PRIMARY KEY or UNIQUE attributes, or by using the DISTINCT option in a query

Retrieval Queries in SQL (2)

- Basic form of the SQL SELECT statement is called a *mapping* or a SELECT-FROM-WHERE *block*

SELECT <attribute list>
FROM <table list>
WHERE <condition>

- <attribute list> is a list of attribute names whose values are to be retrieved by the query
- <table list> is a list of the relation names required to process the query
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query

Where-clause

- **Selection (WHERE)**
  - Boolean expression that holds for all selected tuples in the result
- Comparison and logical operators
  - =, <, <=, >, >=,<>
  - AND, OR, NOT

COMPANY Relational Database Schema

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

Populated Database COMPANY

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
	5	Houston

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22
	Administration	4	987654321	1995-01-01
	Headquarters	1	888665555	1981-06-19

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

Simple SQL Queries

- Basic SQL queries correspond to using the following operations of the relational algebra:
  - SELECT
  - PROJECT
  - JOIN
- All subsequent examples use the COMPANY database

Simple SQL Queries

- **Query 0**: Retrieve the birthdate and address of the employee whose name is 'John B. Smith'.

```
Q0: SELECT BDATE, ADDRESS
    FROM EMPLOYEE
    WHERE FNAME='John' AND MINIT='B'
          AND LNAME='Smith'
```

  - Similar to a SELECT-PROJECT pair of relational algebra operations:
    - The SELECT-clause specifies the projection attributes and the WHERE-clause specifies the selection condition
  - However, the result of the query may contain duplicate tuples

Simple SQL Queries

- **Query 1**: Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1: SELECT FNAME, LNAME, ADDRESS
    FROM EMPLOYEE, DEPARTMENT
    WHERE DNAME='Research' AND DNUMBER=DNO
```

  - Similar to a SELECT-PROJECT-JOIN sequence of relational algebra operations
  - (DNAME='Research') is a selection condition (corresponds to a SELECT operation in relational algebra)
  - (DNUMBER=DNO) is a join condition (corresponds to a JOIN operation in relational algebra)

Simple SQL Queries

- **Query 2**: For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

```
Q2: SELECT PNUMBER, DNUM, LNAME, BDATE, ADDRESS
    FROM PROJECT, DEPARTMENT, EMPLOYEE
    WHERE DNUM=DNUMBER AND MGRSSN=SSN
          AND PLOCATION='Stafford'
```

  - In Q2, there are two join conditions
  - The join condition DNUM=DNUMBER relates a project to its controlling department
  - The join condition MGRSSN=SSN relates the controlling department to the employee who manages that department

Ambiguous Attribute Names

- In SQL, we can use the same name for two (or more) attributes as long as the attributes are in *different relations*
- A query that refers to two or more attributes with the same name must *qualify* the attribute name with the relation name by *prefixing* the relation name to the attribute name
- Example:

  EMPLOYEE.LNAME, **DEPARTMENT.**DNAME

Aliasing – renaming in SQL

- Some queries need to refer to the same relation twice
  - In this case, *aliases* are given to the relation name
- **Query 8**: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

```
Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
    FROM EMPLOYEE AS E, EMPLOYEE AS S
    WHERE E.SUPERSSN = S.SSN
```

  - In Q8, the alternate relation names E and S are called *aliases* or *tuple variables* for the EMPLOYEE relation
  - We can think of E and S as two different *copies* of EMPLOYEE; E represents employees in role of *supervisees* and S represents employees in role of *supervisors*
- Aliasing can also be used in any SQL query for convenience

Unspecified WHERE-clause

- A *missing WHERE-clause* indicates no condition; hence, all tuples of the relations in the FROM-clause are selected
  - This is equivalent to the condition WHERE TRUE

- **Query 9**: Retrieve the SSN values for all employees.

```
Q9: SELECT SSN
    FROM EMPLOYEE
```

Unspecified WHERE-clause

- If more than one relation is specified in the FROM-clause *and* there is no join condition, then the *CARTESIAN PRODUCT* of tuples is selected
- **Query 10**: Retrieve all combinations of EMPLOYEE SSN and DEPARTMENT DNAME values in the database.

```
Q10: SELECT SSN, DNAME
     FROM EMPLOYEE, DEPARTMENT
```

- **It is extremely important not to overlook specifying any selection and join conditions in the WHERE-clause; otherwise, incorrect and very large relations may result**

Use of *

- To retrieve all the attribute values of the selected tuples, a * is used, which stands for ***all the attributes***

- Examples:

```
Q1C:  SELECT *
      FROM EMPLOYEE
      WHERE DNO=5

Q1D:  SELECT *
      FROM EMPLOYEE, DEPARTMENT
      WHERE DNAME='Research' AND DNO=DNUMBER
```

Use of DISTINCT

- SQL does not treat a relation as a set
  - duplicate tuples can appear
- To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used
- For example, the result of Q11 may have duplicate SALARY values whereas Q11A does not have any duplicate values

```
Q11:   SELECT SALARY
       FROM EMPLOYEE

Q11A:  SELECT DISTINCT SALARY
       FROM EMPLOYEE
```

Q11:

Salary
30000
40000
25000
43000
38000
25000
25000
55000

Q11A:

Salary
30000
40000
25000
43000
38000
55000

Set operations

- SQL has directly incorporated some set operations
- There is a union operation (UNION), and in *some versions* of SQL there are set difference (MINUS) and intersection (INTERSECT) operations
  - The resulting relations of these set operations are sets of tuples; *duplicate tuples are eliminated from the result*
  - The set operations apply only to *union compatible relations*; the two relations must have the same attributes and the attributes must appear in the same order

Set operations

- **Query 4**: Make a list of all project numbers for projects that involve an employee whose last name is 'Smith' as a worker on a project or as a manager of the department that controls the project.

```
Q4:  (SELECT PNAME
      FROM PROJECT, DEPARTMENT, EMPLOYEE
      WHERE DNUM=DNUMBER AND MGRSSN=SSN
            AND LNAME='Smith')
     UNION
     (SELECT PNAME
      FROM PROJECT, WORKS_ON, EMPLOYEE
      WHERE PNUMBER=PNO AND ESSN=SSN
            AND LNAME='Smith')
```

Nesting of queries

- A complete SELECT query, called a *nested query*, can be specified within the WHERE-clause of another query, called the *outer query*
  - Many of the previous example queries can be specified in an alternative form using nesting

Query 1: Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1: SELECT FNAME, LNAME, ADDRESS
    FROM EMPLOYEE
    WHERE DNO IN ( SELECT DNUMBER
                   FROM DEPARTMENT
                   WHERE DNAME='Research' )
```

Nesting of queries

- The comparison operator IN compares a value v with a set (or multi-set) of values V, and evaluates to TRUE if v is one of the elements in V
- In general, we can have several levels of nested queries
- A reference to an *unqualified attribute* refers to the relation declared in the ***innermost nested query***
- In this example, the nested query is *not correlated* with the outer query

Correlated nested queries

- If a condition in the WHERE-clause of a *nested query* references an attribute of a relation declared in the *outer query*, the two queries are said to be ***correlated***

- The result of a correlated nested query is different for each tuple (or combination of tuples) of the relation(s) the outer query

- **Query 12**: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
Q12:  SELECT   E.FNAME, E.LNAME
      FROM EMPLOYEE AS E
      WHERE E.SSN IN (SELECT ESSN
                      FROM DEPENDENT
                      WHERE ESSN=E.SSN AND
                            E.FNAME=DEPENDENT_NAME)
```

Correlated nested queries VS join

- A query written with nested SELECT... FROM... WHERE... blocks and using the = or IN comparison operators can ***always*** be expressed as a single block query.
- For example, Q12 may be written as in Q12A

```
Q12A: SELECT E.FNAME, E.LNAME
      FROM EMPLOYEE E, DEPENDENT D
      WHERE E.SSN=D.ESSN AND
            E.FNAME=D.DEPENDENT_NAME
```

The EXISTS function

- EXISTS is used to check whether the result of a correlated nested query is empty (contains no tuples) or not
  - This check is done for each tuple in the outer query
  - If any tuple is retrieved by the nested query the current tuple of the outer query is retrieved in the result
- We can formulate Query 12 in an alternative form that uses EXISTS as Q12B

```
Q12B: SELECT FNAME, LNAME
      FROM EMPLOYEE
      WHERE EXISTS (SELECT *
                    FROM DEPENDENT
                    WHERE SSN=ESSN AND
                          FNAME=DEPENDENT_NAME)
```

The EXISTS function

- **Query 6**: Retrieve the names of employees who have no dependents.

```
Q6: SELECT FNAME, LNAME
    FROM EMPLOYEE
    WHERE NOT EXISTS (SELECT *
                      FROM DEPENDENT
                      WHERE SSN=ESSN)
```

- In Q6, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If ***none exist***, the EMPLOYEE tuple is selected

Explicit sets

- It is also possible to use an **explicit (enumerated) set of values** in the WHERE-clause rather than a nested query
- **Query 13**: Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

```
Q13: SELECT DISTINCT ESSN
     FROM WORKS_ON
     WHERE PNO IN (1, 2, 3)
```

NULL values in SQL queries

- SQL allows queries that check if a value is **NULL** (missing or undefined or not applicable)
- SQL uses **IS** or **IS NOT** to compare NULLs because it considers each NULL value distinct from other NULL values, so *equality comparison is not appropriate*.
- **Query 14**: Retrieve the names of all employees who do not have supervisors.

```
Q14: SELECT FNAME, LNAME
     FROM EMPLOYEE
     WHERE SUPERSSN IS NULL
```

If a join condition is specified, tuples with NULL values for the join attributes are not included in the result

Joined Relations Feature

- Can specify a "joined relation" in the FROM-clause
  - Looks like any other relation but is the result of a join
  - Allows the user to specify different types of joins
    - regular "theta" JOIN
    - NATURAL JOIN
    - LEFT OUTER JOIN
    - RIGHT OUTER JOIN
    - CROSS JOIN, etc

Joined Relations Feature

- **Query 8**: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

```
Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
    FROM EMPLOYEE AS E,EMPLOYEE AS S
    WHERE E.SUPERSSN=S.SSN
```

- Can be written as:

```
Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
    FROM (EMPLOYEE AS E
          LEFT OUTER JOIN
          EMPLOYEE AS S
          ON E.SUPERSSN=S.SSN)
```

Joined Relations Feature

```
Q1:  SELECT FNAME, LNAME, ADDRESS
     FROM EMPLOYEE, DEPARTMENT
     WHERE DNAME='Research' AND DNUMBER=DNO
```

- Can be written as:

```
Q1*:  SELECT FNAME, LNAME, ADDRESS
      FROM (EMPLOYEE
            JOIN DEPARTMENT ON DNUMBER=DNO)
      WHERE DNAME='Research'
```

- Or as:

```
Q1**:  SELECT FNAME, LNAME, ADDRESS
       FROM (EMPLOYEE
             NATURAL JOIN
             (DEPARTMENT AS
              DEPT(DNAME, DNO, MSSN, MSDATE)))
       WHERE DNAME='Research'
```

It is important to use renaming of DNUMBER attribute so a NATURAL JOIN would be possible

Joined Relations Feature

- An example for multiple joins in the joined tables
- Q2 could be written as:

```
Q2: SELECT PNUMBER,DNUM,LNAME,BDATE,ADDRESS
    FROM ((PROJECT JOIN
           DEPARTMENT ON DNUM=DNUMBER)
           JOIN EMPLOYEE ON MGRSSN=SSN))
    WHERE PLOCATION='Stafford'
```

Aggregate functions

- **Aggregate functions** are used to summarize information from multiple tuples into a single-tuple summary
  - Include: **COUNT, SUM, MAX, MIN,** and **AVG**.
  - The result is always a single value

- **Query 15**: Find the maximum salary, the minimum salary, and the average salary among all employees.

```
Q15:
    SELECT MAX(SALARY), MIN(SALARY), AVG(SALARY)
    FROM EMPLOYEE
```

Aggregate functions

- **Query 16**: Find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.

```
Q16:
    SELECT MAX(SALARY), MIN(SALARY), AVG(SALARY)
    FROM EMPLOYEE, DEPARTMENT
    WHERE DNO=DNUMBER AND DNAME='Research'
```

Aggregate functions

- **Queries 17 and 18**: Retrieve the total number of employees in the company (Q17), and the number of employees in the 'Research' department (Q18).

```
Q17:  SELECT COUNT(*)
      FROM EMPLOYEE

Q18:  SELECT COUNT(*)
      FROM EMPLOYEE, DEPARTMENT
      WHERE DNO=DNUMBER AND DNAME='Research'
```

Difference between COUNT(*) and COUNT(attribute)

- The asterix (*) denotes the rows (tuples), hence COUNT (*) results in the number of rows.
- We can use the COUNT function to count values in a column rather than a tuple
- **Query 23:** Count the number of distinct salary values in the database.

```
Q23: SELECT COUNT (DISTINCT Salary)
     FROM Employee;
```

- If we write COUNT(Salary) instead of COUNT (DISTINCT Salary) in Q23, then the duplicate values will not be removed. However, any tuples with NULL for SALARY will not be counted.
- In general, NULL values are **discarded when aggregate functions** are applied to a particular column (attribute).

Grouping

- In many cases, we want to apply the aggregate functions to ***subgroups of tuples*** in a relation
- Each subgroup of tuples consists of the set of tuples that have the *same value* for the *grouping attribute(s)*
- The function is applied to each subgroup independently
- SQL has a **GROUP BY**-clause for specifying the grouping attributes, which *must also appear in the SELECT-clause*

Grouping

- **Query 20**: For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q20: SELECT DNO, COUNT (*), AVG (SALARY)
     FROM EMPLOYEE
     GROUP BY DNO
```

- In Q20, the EMPLOYEE tuples are divided into groups
  - Each group having the same value for the grouping attribute DNO
- The COUNT and AVG functions are applied to each such group of tuples separately
- The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples

Grouping

- A join condition can be used in conjunction with grouping
- **Query 21**: For each project, retrieve the project number, project name, and the number of employees who work on that project.

```
Q21: SELECT PNUMBER, PNAME, COUNT(*)
     FROM PROJECT, WORKS_ON
     WHERE PNUMBER=PNO
     GROUP BY PNUMBER, PNAME
```

- In this case, the grouping and functions are applied after the joining of the two relations

The HAVING-clause

- Sometimes we want to retrieve the values of these functions for only those ***groups that satisfy certain conditions***
- The **HAVING**-clause is used for specifying a selection condition on groups (rather than on individual tuples)

The HAVING-clause

- **Query 22**: For each project *on which more than two employees work*, retrieve the project number, project name, and the number of employees who work on that project.

```
Q22: SELECT PNUMBER, PNAME, COUNT(*)
     FROM PROJECT, WORKS_ON
     WHERE PNUMBER=PNO
     GROUP BY PNUMBER, PNAME
     HAVING COUNT(*) > 2
```

Substring comparison

- The **LIKE** comparison operator is used to compare partial strings
- Two reserved characters are used:
  - '**%**' (or '*' in some implementations) replaces an arbitrary number of characters, and
  - '_' (or '?')replaces a single arbitrary character
- **Query 25**: Retrieve all employees whose address is in Houston, Texas.
  - Here, the value of the ADDRESS attribute must contain the substring 'Houston,TX' in it.

```
Q25: SELECT FNAME, LNAME
     FROM EMPLOYEE
     WHERE ADDRESS LIKE '%Houston,TX%'
```

Substring comparison

- **Query 26**: Retrieve all employees who were born during the 1950s.
  - Here, '5' must be the 8th character of the string (according to our format for date), so the BDATE value is '_______5_', with each underscore as a place holder for a single arbitrary character.

```
Q26: SELECT FNAME, LNAME
     FROM EMPLOYEE
     WHERE BDATE LIKE '_______5_'
```

Arithmetic operations

- The standard arithmetic operators **'+', '-'. '*', and '/'** (for addition, subtraction, multiplication, and division, respectively) can be applied to numeric values in an SQL query result

- **Query 27**: Show the effect of giving all employees who work on the 'ProductX' project a 10% raise.

```
Q27: SELECT FNAME, LNAME, 1.1*SALARY
     FROM EMPLOYEE, WORKS_ON, PROJECT
     WHERE SSN=ESSN AND PNO=PNUMBER
           AND PNAME='ProductX'
```

ORDER BY

- The **ORDER BY** clause is used to sort the tuples in a query result based on the values of some attribute(s)
- **Query 28**: Retrieve a list of employees and the projects each works in, ordered by the employee's department, and within each department ordered alphabetically by employee last name

```
Q28: SELECT DNAME, LNAME, FNAME, PNAME
     FROM DEPARTMENT,EMPLOYEE,WORKS_ON,PROJECT
     WHERE DNUMBER=DNO AND SSN=ESSN
           AND PNO=PNUMBER
     ORDER BY DNAME, LNAME
```

ORDER BY

- The default order is in ascending order of values
- We can specify the keyword **DESC** if we want a descending order
- The keyword **ASC** can be used to explicitly specify ascending order, even though it is the default

Summary of SQL Queries

- A query in SQL can consist of up to six clauses, but only the first two, SELECT and FROM, are mandatory. The clauses are specified in the following order:

```
SELECT      <attribute list>
FROM        <table list>
[WHERE      <condition>]
[GROUP BY   <grouping attribute(s)>]
[HAVING     <group condition>]
[ORDER BY   <attribute list>]
```

Specifying Updates in SQL

- There are three SQL commands to modify the database:
  - **INSERT**
  - **DELETE** and
  - **UPDATE**

INSERT

INSERT INTO <table name> **VALUES** (<atributes>)

- In its simplest form, it is used to add one or more tuples to a relation
  - Attribute values should be listed in the same order as the attributes were specified in the **CREATE TABLE** command
- An alternate form of INSERT specifies explicitly the attribute names that correspond to the values in the new tuple
  - Attributes with NULL values can be left out

INSERT

- Example:

```
U1: INSERT INTO EMPLOYEE VALUES
    ('Richard','K','Marini',
     '653298653', '30-DEC-52',
     '98 Oak Forest,Katy,TX', 'M', 37000,
     '987654321', 4)
```

- Example: Insert a tuple for a new EMPLOYEE for whom we only know the FNAME, LNAME, and SSN attributes.

```
U1A: INSERT INTO EMPLOYEE (FNAME, LNAME, SSN)
     VALUES ('Richard', 'Marini', '653298653')
```

INSERT

- Another variation of INSERT allows insertion of *multiple tuples* resulting from a query into a relation
- Example: Suppose we want to create a temporary table that has the name, number of employees, and total salaries for each department.

```
U3A: CREATE TABLE DEPTS_INFO
     (DEPT_NAME VARCHAR(10),
      NO_OF_EMPS INTEGER,
      TOTAL_SAL INTEGER);

U3B: INSERT INTO
     DEPTS_INFO (DEPT_NAME, NO_OF_EMPS, TOTAL_SAL)
     SELECT DNAME, COUNT(*), SUM(SALARY)
     FROM DEPARTMENT, EMPLOYEE
     WHERE DNUMBER=DNO
     GROUP BY DNAME ;
```

INSERT

- Notes:
  - A table DEPTS_INFO is created by U3A, and is loaded with the summary information retrieved from the database by the query in U3B
  - The DEPTS_INFO table may not be up-to-date if we change the tuples in either the DEPARTMENT or the EMPLOYEE relations *after* issuing U3B.
  - We have to create a view (see later slides) to keep such a table up to date.

DELETE

DELETE FROM <table> **WHERE** <conditions>

- Removes tuples from a relation
  - Includes a WHERE-clause to select the tuples to be deleted
    - A missing WHERE-clause specifies that *all tuples* in the relation are to be deleted; the table then becomes an empty table
    - The number of tuples deleted depends on the number of tuples in the relation that satisfy the WHERE-clause
  - Referential integrity should be enforced
  - Tuples are deleted from only *one table* at a time (unless CASCADE is specified on a referential integrity constraint)

DELETE

- Examples:

```
U4A: DELETE FROM EMPLOYEE
     WHERE LNAME='Brown'

U4B: DELETE FROM EMPLOYEE
     WHERE SSN='123456789'

U4C: DELETE FROM EMPLOYEE
     WHERE DNO IN (SELECT DNUMBER
                   FROM DEPARTMENT
                   WHERE DNAME='Research')

U4D: DELETE FROM EMPLOYEE
```

UPDATE

UPDATE <table> **SET** <attributes>
WHERE <conditions>

- Used to modify attribute values of one or more selected tuples
- A WHERE-clause selects the tuples to be modified
- An additional SET-clause specifies the attributes to be modified and their new values
- Each command modifies tuples *in the same relation*
- Referential integrity should be enforced

UPDATE

- Example: Change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

```
U5: UPDATE PROJECT
    SET PLOCATION = 'Bellaire', DNUM = 5
    WHERE PNUMBER = 10
```

UPDATE

- Example: Give all employees in the 'Research' department a 10% raise in salary.

```
U6: UPDATE EMPLOYEE
    SET SALARY = SALARY *1.1
    WHERE DNO IN (SELECT DNUMBER
                  FROM DEPARTMENT
                  WHERE DNAME='Research')
```

- In this request, the modified SALARY value depends on the original SALARY value in each tuple
  - The reference to the SALARY attribute on the **right** of = refers to the **old** SALARY value before modification
  - The reference to the SALARY attribute on the **left** of = refers to the **new** SALARY value after modification

Introduction to SQL Programming Techniques

- General Constraints as Assertions
- Views in SQL
- Database Programming
- Embedded SQL
- Functions Calls, SQL/CLI
- Stored Procedures, SQL/PSM

Constraints as Assertions

- In SQL, users can specify general constraints—those that do not fall into any of the categories described in previous slides—via **declarative assertions, using the CREATE ASSERTION statement of the DDL**
- **`CREATE ASSERTION`** `<name>` **`CHECK`** `(condition)`
  - Components include:
    - a constraint name,
    - followed by **`CHECK`**,
    - followed by a condition

Assertions: An Example

- "The salary of an employee must not be greater than the salary of the manager of the department that the employee works for''

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS
       (SELECT *
        FROM EMPLOYEE E, EMPLOYEE M,DEPARTMENT D
        WHERE E.SALARY > M.SALARY AND
              E.DNO = D.NUMBER AND D.MGRSSN = M.SSN))
```

- Specify a query that violates the condition; include inside a `NOT EXISTS` clause
- Query result must be empty
  - if the query result is not empty, the assertion has been violated

SQL Triggers

- Objective: **to monitor** a database and take initiate action when a condition occurs
- Triggers are expressed in a syntax similar to assertions and include the following:
  - **Event**
    - Such as an insert, delete, or update operation
  - **Condition**
  - **Action**
    - To be taken when the condition is satisfied

SQL Triggers

```
CREATE TRIGGER <trigger name>
(AFTER | BEFORE) <events> ON <table>
[FOR EACH ROW]
[WHEN <conditions>]
    <actions>;
```

```
<events>::=<event> {OR <event>}
<event>::=INSERT|DELETE|UPDATE [OF <attribute> {, <attribute>}]
<actions>::=<PL/SQL block>
```

SQL Triggers: An Example

- A trigger to compare an employee's salary to his/her supervisor during insert or update operations

```
CREATE TRIGGER INFORM_SUPERVISOR
BEFORE INSERT OR UPDATE OF
       SALARY, SUPERVISOR_SSN ON EMPLOYEE
FOR EACH ROW
WHEN
(NEW.SALARY > (SELECT SALARY
               FROM EMPLOYEE
               WHERE SSN=NEW.SUPERVISOR_SSN)
)
INFORM_SUPERVISOR (NEW.SUPERVISOR_SSN, NEW.SSN);
```

This is some stored procedure

SQL Triggers

- **HOMEWORK**:
  Add a new attribute to the DEPARTMENT table and name it *totalSalary.* This attribute should store the total salary of all employees of the department.

- Write the triggers for the following events:
  - Insertion of new employees in the database
  - Update of the salary of existing employees
  - Reassignment of existing employees from one department to another
  - Deletion of employees from the database

Views in SQL

- A view is a “virtual” table that is derived from other tables
- Allows for limited update operations
  - Since the table may not physically be stored
- Allows full query operations
- A convenience for expressing certain operations

Specification of Views

```
CREATE VIEW <name> [(<attribute>{,<attribute>})]
                AS <SELECT statement>
```

- View specification includes:
  - a table (view) name
  - a possible list of attribute names (for example, when arithmetic operations are specified or when we want the names to be different from the attributes in the base relations)
  - a query to specify the table contents

SQL Views: An Example

- Specify a different WORKS_ON table (*WORKS_ON_NEW*) that will swap the employee SSN with his/her first name and last name

V1:

```
CREATE VIEW WORKS_ON_NEW AS
SELECT FNAME, LNAME, PNAME, HOURS
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE SSN=ESSN AND PNO=PNUMBER;
```

Using a Virtual Table

- We can specify SQL queries on the newly created virtual table (view)

```
SELECT FNAME, LNAME
FROM WORKS_ON_NEW
WHERE PNAME='ProductX';
```

- When no longer needed, a view can be dropped

```
DROP WORKS_ON_NEW;
```

Efficient View Implementation

- **Query modification**:
  - Present the view query in terms of a query on the underlying base tables

```
SELECT FNAME, LNAME
FROM WORKS_ON_NEW
WHERE PNAME='ProductX';
```

```
SELECT FNAME, LNAME
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE SSN=ESSN AND PNO=PNUMBER
      AND PNAME = 'ProductX'
```

- Disadvantage: Inefficient for views defined via complex queries
  - Especially if additional queries are to be applied to the view within a short time period

Efficient View Implementation

- **View materialization**:
  - Involves physically creating and keeping a temporary table
  - Assumption:
    - Other queries on the view will follow
  - Concerns:
    - Maintaining correspondence between the base table and the view when the base table is updated
  - Strategy:
    - Incremental update

Update Views

- Update on a single view without aggregate operations:
  - Update may map to an update on the underlying base table
- Views involving joins:
  - An update *may* map to an update on the underlying base relations
    - Not always possible

Un-updatable Views

- Views defined using groups and aggregate functions are **not updateable**
- Views defined on multiple tables using joins are generally not updateable
- **`WITH CHECK OPTION`**: must be added to the definition of a view if the view is to be updated
  - To allow check for updatability and to plan for an execution strategy

Database Programming

- Objective:
  - To access a database from an application program (as opposed to interactive interfaces)
- Why?
  - An interactive interface is convenient but not sufficient
    - A majority of database operations are made thru application programs (increasingly thru web applications)

Database Programming Approaches

- Embedded commands:
  - Database commands are embedded in a general-purpose programming language
- Library of database functions:
  - Available to the host language for database calls; known as an *API*
    - *API* standards for Application Program Interface
- A brand new, full-fledged language
  - Minimizes impedance mismatch

Impedance Mismatch

- Incompatibilities between a host programming language and the database model, e.g.,
  - type mismatch and incompatibilities; requires a new binding for each language
  - set vs. record-at-a-time processing
    - need special iterators to loop over query results and manipulate individual values

Steps in Database Programming

1. Client program ***opens a connection*** to the database server
2. Client program ***submits queries to and/or updates*** the database
3. When database access is no longer needed, client program ***closes (terminates) the connection***

SQL Commands for Connecting to a Database

- Connection (multiple connections are possible but only one is active)

```
CONNECT TO server-name AS connection-name
AUTHORIZATION user-account-info;
```

- Change from an active connection to another one

```
SET CONNECTION connection-name;
```

- Disconnection

```
DISCONNECT connection-name;
```

Embedded SQL

- Most SQL statements can be embedded in a general-purpose *host* programming language such as COBOL, C, Java
- An embedded SQL statement is distinguished from the host language statements by enclosing it between **`EXEC SQL`** or **`EXEC SQL BEGIN`** and a matching **`END-EXEC`** or **`EXEC SQL END`** (or semicolon)
  - Syntax may vary with language
  - *Shared variables* (used in both languages) usually prefixed with a colon (:) in SQL

Example: Variable Declaration in C

- Variables inside **DECLARE** are shared and can appear (while prefixed by a colon) in SQL statements
- **SQLCODE** is used to communicate errors/exceptions between the database and the program

```
int loop;
EXEC SQL BEGIN DECLARE SECTION;
    varchar dname[16], fname[16], …;
    char ssn[10], bdate[11], …;
    int dno, dnumber, SQLCODE, …;
EXEC SQL END DECLARE SECTION;
```

Embedded SQL in C Programming Examples

```
loop = 1;
while (loop) {
    prompt ("Enter SSN: ", ssn);
    EXEC SQL
        select FNAME, LNAME, ADDRESS, SALARY
        into :fname, :lname, :address, :salary
        from EMPLOYEE where SSN == :ssn;
        if (SQLCODE == 0) printf(fname, …);
        else printf("SSN does not exist: ", ssn);
        prompt("More SSN? (1=yes, 0=no): ", loop);
    END-EXEC
}
```

Embedded SQL in C Programming Examples

- A **cursor** (iterator) is needed to process multiple tuples
- **`FETCH`** commands move the cursor to the *next* tuple
- **`CLOSE CURSOR`** indicates that the processing of query results has been completed

C program segment that uses cursors with embedded SQL for update purposes

```
0) prompt("Enter the Department Name: ", dname) ;
1) EXEC SQL
2) select Dnumber into :dnumber
3) from DEPARTMENT where Dname = :dname ;
4) EXEC SQL DECLARE EMP CURSOR FOR
5) select Ssn, Fname, Minit, Lname, Salary
6) from EMPLOYEE where Dno = :dnumber
7) FOR UPDATE OF Salary ;
8) EXEC SQL OPEN EMP ;
9) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE == 0) {
11) printf("Employee name is:", Fname, Minit, Lname) ;
12) prompt("Enter the raise amount: ", raise) ;
13) EXEC SQL
14) update EMPLOYEE
15) set Salary = Salary + :raise
16) where CURRENT OF EMP ;
17) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
18) }
19) EXEC SQL CLOSE EMP ;
```

Dynamic SQL

- Objective:
  - Composing and executing new (not previously compiled) SQL statements at run-time
    - a program accepts SQL statements from the keyboard at run-time
    - a point-and-click operation translates to certain SQL query
- Dynamic update is relatively simple; dynamic query can be complex
  - because the type and number of retrieved attributes are unknown at compile time

Dynamic SQL

```
EXEC SQL BEGIN DECLARE SECTION;
varchar sqlupdatestring[256];
EXEC SQL END DECLARE SECTION;
…
prompt ("Enter update command:", sqlupdatestring);
EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring;
EXEC SQL EXECUTE sqlcommand;
```

Embedded SQL in Java

- SQLJ: a standard for embedding SQL in Java
- An SQLJ translator converts SQL statements into Java
  - These are executed thru the *JDBC* interface
- Certain classes have to be imported
  - E.g., **`java.sql`**

Embedded SQL in Java: An Example

```
ssn = readEntry("Enter a SSN: ");
try {
    #sql{select FNAME, LNAME, ADDRESS, SALARY
    into :fname, :lname, :address, :salary
    from EMPLOYEE where SSN = :ssn};
}
catch (SQLException se) {
    System.out.println("SSN does not exist: ",+ssn);
    return;
}
System.out.println(fname + " " + lname + … );
```

Java Database Connectivity (JDBC)

- JDBC:
  - SQL connection function calls for Java programming
- A Java program with JDBC functions can access any relational DBMS that has a JDBC driver
- JDBC allows a program to connect to several databases (known as *data sources*)

Steps in JDBC Database Access

1. Import JDBC library **`(java.sql.*)`**
2. Load JDBC driver: **`Class.forname("oracle.jdbc.driver.OracleDriver")`**
3. Define appropriate variables
4. Create a connect object (via **`getConnection`**)
5. Create a statement object from the **`Statement`** class:
   1. **`PreparedStatment`** 2. **`CallableStatement`**
6. Identify statement parameters (designated by question marks)
7. Bound parameters to program variables
8. Execute SQL statement (referenced by an object) via JDBC's **`execute`**
   1. **`executeUpdate`** 2. **`executeQuery`**
6. Process query results (returned in an object of type **`ResultSet`**)

➚ **`ResultSet`** is a 2-dimentional table in the example

Java program segment with JDBC

```
0) import java.io.* ;
1) import java.sql.*
...
2) class getEmpInfo {
3) public static void main (String args []) throws SQLException, IOException {
4) try { Class.forName("oracle.jdbc.driver.OracleDriver")
5) } catch (ClassNotFoundException x) {
6) System.out.println ("Driver could not be loaded") ;
7) }
8) String dbacct, passwrd, ssn, lname ;
9) Double salary ;
10) dbacct = readentry("Enter database account:") ;
11) passwrd = readentry("Enter password:") ;
12) Connection conn = DriverManager.getConnection
13) ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
14) String stmt1 = "select Lname, Salary from EMPLOYEE where Ssn = ?" ;
15) PreparedStatement p = conn.prepareStatement(stmt1) ;
16) ssn = readentry("Enter a Social Security Number: ") ;
17) p.clearParameters() ;
18) p.setString(1, ssn) ;
19) ResultSet r = p.executeQuery() ;
20) while (r.next()) {
21) lname = r.getString(1) ;
22) salary = r.getDouble(2) ;
23) system.out.printline(lname + salary) ;
24) } }
25) }
```

Database Stored Procedures

- Persistent procedures/functions (modules) are stored locally and executed by the database server
  - As opposed to execution by clients
- Advantages:
  - If the procedure is needed by many applications, it can be invoked by any of them (thus reduce duplications)
  - Execution by the server reduces communication costs
  - Enhance the modeling power of views
- Disadvantages:
  - Every DBMS has its own syntax and this can make the system less portable

Stored Procedure Constructs

- Stored procedure

```
CREATE PROCEDURE procedure-name (params)
local-declarations
procedure-body;
```

- Stored function

```
CREATE FUNCTION fun-name (params) RETRUNS return-type
local-declarations
function-body;
```

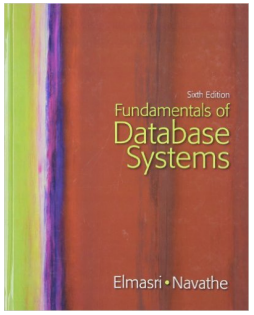
- Calling a procedure of function

```
CALL procedure-name/fun-name (arguments);
```

Short summary

- Assertions provide a means to specify additional constraints
- Triggers are assertions that define actions to be automatically taken when certain conditions occur
- Views create temporary (virtual) tables
- A database may be accessed in an interactive mode
- Most often, however, data in a database is manipulate via application programs
- Several methods of database programming:
  - Embedded SQL
  - Dynamic SQL
  - Stored procedure and function

Bibliography



- Chapter 4
- Chapter 5
- Chapter 13

- Chapter 6
- Chapter 9