Database Systems

Session 6 – Main Theme

Standard Query Language (SQL) Features

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

1. Session Overview
2. Basic SQL
3. Advanced SQL
4. Introduction to SQL Programming
5. Web Database Programming Using PhP
6. Summary and Conclusion

Session Agenda

- Session Overview
- Basic SQL
- Data Manipulation Language
- Summary & Conclusion

What is the class about?

- Course description and syllabus:
  - http://www.nyu.edu/classes/jcf/CSCI-GA.2433-001
  - http://cs.nyu.edu/courses/spring16/CSCI-GA.2433-001/
- Textbooks:
  - ***Fundamentals of Database Systems (7th Edition)***

Ramez Elmasri and Shamkant Navathe
Pearson
ISBN-10: 0133970779, ISBN-13: 978-0133970777 7th Edition (06/18/15)

Icons / Metaphors

Information

Common Realization

Knowledge/Competency Pattern

Governance

Alignment

Solution Approach

Agenda

1. Session Overview
2. Basic SQL
3. Advanced SQL
4. Introduction to SQL Programming
5. Web Database Programming Using PhP
6. Summary and Conclusion

Agenda

- SQL Data Definition and Data Types
- Specifying Constraints in SQL
- Basic Retrieval Queries in SQL
- `INSERT`, `DELETE`, and `UPDATE` Statements in SQL
- Additional Features of SQL

- SQL language
  - Considered one of the major reasons for the commercial success of relational databases
- SQL
  - The origin of SQL is relational predicate calculus called tuple calculus which was proposed initially as the language SQUARE.
  - SQL Actually comes from the word “SEQUEL” which was the original term used in the paper: “SEQUEL TO SQUARE” by Chamberlin and Boyce. IBM could not copyright that term, so they abbreviated to SQL and copyrighted the term SQL.
  - Now popularly known as “Structured Query language”.
  - SQL is an informal or practical rendering of the relational data model with syntax

SQL Data Definition, Data Types, Standards

- Terminology:
  - **Table**, **row**, and **column** used for relational model terms relation, tuple, and attribute
- `CREATE` statement
  - Main SQL command for data definition
- The language has features for : Data definition, Data Manipulation, Transaction control (Transact-SQL), Indexing, Security specification (Grant and Revoke), Active databases, Multi-media, Distributed databases, etc.

SQL Standards

- SQL has gone through many standards: starting with SQL-86 or SQL 1.A. SQL-92 is referred to as SQL-2.
- Later standards (from SQL-1999) are divided into **core** specification and specialized **extensions**. The extensions are implemented for different applications – such as data mining, data warehousing, multimedia etc.
- SQL-2006 added XML features; In 2008 they added Object-oriented features.
- SQL-3 is the current standard which started with SQL-1999. It is not fully implemented in any RDBMS.

Schema and Catalog Concepts in SQL (1/2)

- We cover the basic standard SQL syntax – there are variations in existing RDBMS systems
- **SQL schema**
  - Identified by a **schema name**
  - Includes an **authorization identifier** and **descriptors** for each element
- **Schema elements** include
  - Tables, constraints, views, domains, and other constructs
- Each statement in SQL ends with a **semicolon**

Schema and Catalog Concepts in SQL (2/2)

- `CREATE SCHEMA` statement
  - `CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';`
- **Catalog**
  - Named collection of schemas in an SQL environment
- SQL also has the concept of a cluster of catalogs.

The CREATE TABLE Command in SQL (1/5)

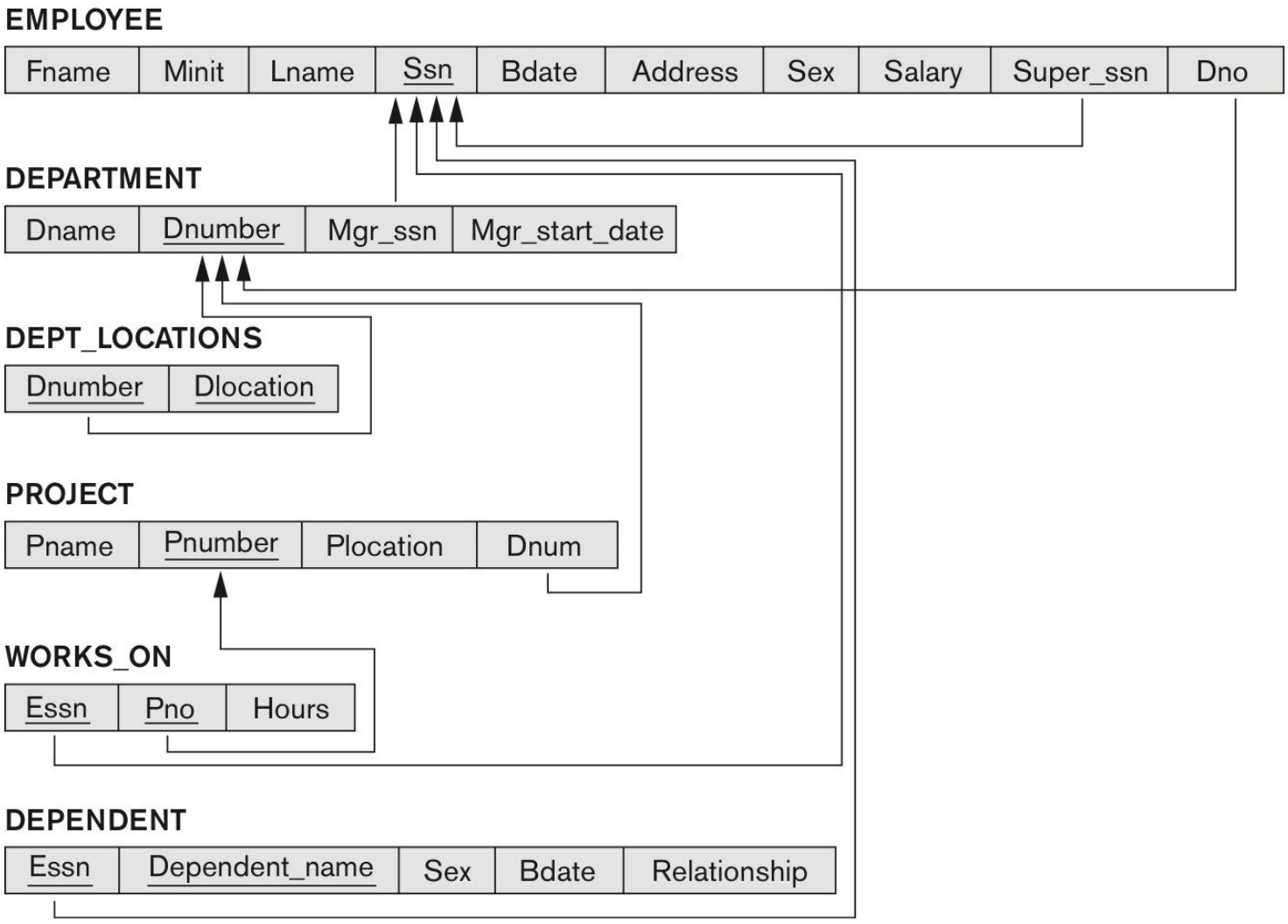
- Specifying a new relation
  - Provide name of table
  - Specify attributes, their types and initial constraints
- Can optionally specify schema:
  - `CREATE TABLE COMPANY.EMPLOYEE ...`

    or
  - `CREATE TABLE EMPLOYEE ...`

The CREATE TABLE Command in SQL (2/5)

- **Base tables** (**base relations**)
  - Relation and its tuples are actually created and stored as a file by the DBMS
- **Virtual relations (views)**
  - Created through the `CREATE VIEW` statement. Do not correspond to any physical file.

Company Relational Database



One Possible Database State for COMPANY RDB Schema (1/2)

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	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	1968-01-19	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

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

One Possible Database State for COMPANY RDB Schema (2/2)

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

SQL CREATE TABLE Data Definition Statements (Company Schema)

```
CREATE TABLE EMPLOYEE
        ( Fname                 VARCHAR(15)             NOT NULL,
          Minit                 CHAR,
          Lname                 VARCHAR(15)             NOT NULL,
          Ssn                   CHAR(9)                 NOT NULL,
          Bdate                 DATE,
          Address               VARCHAR(30),
          Sex                   CHAR,
          Salary                DECIMAL(10,2),
          Super_ssn             CHAR(9),
          Dno                   INT                     NOT NULL,
        PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)             NOT NULL,
          Dnumber               INT                     NOT NULL,
          Mgr_ssn               CHAR(9)                 NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                     NOT NULL,
          Dlocation             VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
```

SQL CREATE TABLE Data Definition Statements (Company Schema)

```
CREATE TABLE PROJECT
        ( Pname                 VARCHAR(15)             NOT NULL,
          Pnumber               INT                     NOT NULL,
          Plocation             VARCHAR(15),
          Dnum                  INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn                  CHAR(9)                 NOT NULL,
          Pno                   INT                     NOT NULL,
          Hours                 DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn                  CHAR(9)                 NOT NULL,
          Dependent_name        VARCHAR(15)             NOT NULL,
          Sex                   CHAR,
          Bdate                 DATE,
          Relationship          VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

The CREATE TABLE Command in SQL (3/5)

- Some foreign keys may cause errors
  - Specified either via:
    - Circular references
    - Or because they refer to a table that has not yet been created
  - DBA's have ways to stop referential integrity enforcement to get around this problem.

Attribute Data Types and Domains in SQL (1/4)

- Basic **data types**
  - **Numeric** data types
    - Integer numbers: `INTEGER`, `INT`, and `SMALLINT`
    - Floating-point (real) numbers: `FLOAT` or `REAL`, and `DOUBLE PRECISION`
  - **Character-string** data types
    - Fixed length: `CHAR(`*n*`)`, `CHARACTER(`*n*`)`
    - Varying length: `VARCHAR(`*n*`)`, `CHAR VARYING(`*n*`)`, `CHARACTER VARYING(`*n*`)`

Attribute Data Types and Domains in SQL (2/4)

- **Bit-string** data types
  - Fixed length: `BIT(`*n*`)`
  - Varying length: `BIT VARYING(`*n*`)`
- **Boolean** data type
  - Values of `TRUE` or `FALSE` or `NULL`
- **DATE** data type
  - Ten positions
  - Components are `YEAR`, `MONTH`, and `DAY` in the form YYYY-MM-DD
  - Multiple mapping functions available in RDBMSs to change date formats

Attribute Data Types and Domains in SQL (3/4)

- Additional data types
  - **Timestamp** data type

    Includes the `DATE` and `TIME` fields
    - Plus a minimum of six positions for decimal fractions of seconds
    - Optional `WITH TIME ZONE` qualifier
  - **INTERVAL** data type
    - Specifies a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp
  - **DATE, TIME, Timestamp, INTERVAL** data types can be **cast** or converted to string formats for comparison.

Attribute Data Types and Domains in SQL (4/4)

- **Domain**
  - Name used with the attribute specification
  - Makes it easier to change the data type for a domain that is used by numerous attributes
  - Improves schema readability
  - Example:
    - `CREATE DOMAIN SSN_TYPE AS CHAR(9);`
- **TYPE**
  - User Defined Types (UDTs) are supported for object-oriented applications. Uses the command: `CREATE TYPE`

Basic constraints:

- Relational Model has 3 basic constraint types that are supported in SQL:
  - **Key** constraint: A primary key value cannot be duplicated
  - **Entity Integrity** Constraint: A primary key value cannot be null
  - **Referential integrity** constraints : The “foreign key “ must have a value that is already present as a primary key, or may be null.

Specifying Attribute Constraints

Other Restrictions on attribute domains:

- Default value of an attribute
  - **DEFAULT** <value>
  - NULL is not permitted for a particular attribute (NOT NULL)
- **CHECK** clause
  - `Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber < 21);`

Specifying Key and Referential Integrity Constraints (1/3)

- **`PRIMARY KEY`** clause
  - Specifies one or more attributes that make up the primary key of a relation
  - `Dnumber INT PRIMARY KEY;`
- **`UNIQUE`** clause
  - Specifies alternate (secondary) keys (called CANDIDATE keys in the relational model).
  - `Dname VARCHAR(15) UNIQUE;`

Specifying Key and Referential Integrity Constraints (2/3)

- **`FOREIGN KEY`** clause
  - Default operation: reject update on violation
  - Attach **referential triggered action** clause
    - Options include `SET NULL`, `CASCADE`, and `SET DEFAULT`
    - Action taken by the DBMS for `SET NULL` or `SET DEFAULT` is the same for both `ON DELETE` and `ON UPDATE`
    - `CASCADE` option suitable for “relationship” relations

Giving Names to Constraints

- Using the Keyword **`CONSTRAINT`**
  - Name a constraint
  - Useful for later altering

Default Attribute Values and Ref. Integrity Triggered Action Spec.

```
CREATE TABLE EMPLOYEE
    ( … ,
      Dno          INT          NOT NULL       DEFAULT 1,
    CONSTRAINT EMPPK
      PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
      FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                  ON DELETE SET NULL          ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
      FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                  ON DELETE SET DEFAULT       ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
    ( … ,
      Mgr_ssn CHAR(9)          NOT NULL       DEFAULT '888665555',
      … ,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                  ON DELETE SET DEFAULT       ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
    ( … ,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                  ON DELETE CASCADE           ON UPDATE CASCADE);
```

Specifying Constraints on Tuples Using CHECK

- Additional Constraints on individual tuples within a relation are also possible using CHECK
- `CHECK` clauses at the end of a `CREATE TABLE` statement
  - Apply to each tuple individually
  - `CHECK (Dept_create_date <= Mgr_start_date);`

Basic Retrieval Queries in SQL

- `SELECT` statement
  - One basic statement for retrieving information from a database
- SQL allows a table to have two or more tuples that are identical in all their attribute values
  - Unlike relational model (relational model is strictly set-theory based)
  - Multiset or bag behavior
  - Tuple-id may be used as a key

The SELECT-FROM-WHERE Structure of Basic SQL Queries (1/4)

- Basic form of the `SELECT` statement:

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

where

- <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.

The SELECT-FROM-WHERE Structure of Basic SQL Queries (2/4)

- Logical comparison operators
  - =, <, <=, >, >=, and <>
- **Projection attributes**
  - Attributes whose values are to be retrieved
- **Selection condition**
  - Boolean condition that must be true for any retrieved tuple. Selection conditions include join conditions when multiple relations are involved.

Basic Retrieval Queries (1/2)

Bdate	Address
1965-01-09	731Fondren, Houston, TX

Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:   SELECT   Bdate, Address
      FROM     EMPLOYEE
      WHERE    Fname='John' AND Minit='B' AND Lname='Smith';
```

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;
```

Basic Retrieval Queries (2/2)

(c)

Pnumber	Dnum	Lname	Address	Bdate
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

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 birth date.

```
Q2:   SELECT   Pnumber, Dnum, Lname, Address, Bdate
      FROM     PROJECT, DEPARTMENT, EMPLOYEE
      WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn AND
               Plocation='Stafford';
```

Ambiguous Attribute Names

- Same name can be used for two (or more) attributes in different relations
  - As long as the attributes are in different relations
  - Must **qualify** the attribute name with the relation name to prevent ambiguity

```
Q1A:  SELECT  Fname, EMPLOYEE.Name, Address
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DEPARTMENT.Name='Research' AND
              DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;
```

Aliasing, Renaming, and Tuple Variables (1/2)

- **Aliases** or **tuple variables**
  - Declare alternative relation names E and S to refer to the EMPLOYEE relation twice in a query:

Query 8. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

- ```
  SELECT  E.Fname, E.Lname, S.Fname, S.Lname
  FROM  EMPLOYEE AS E, EMPLOYEE AS S
  WHERE E.Super_ssn=S.Ssn;
  ```
  - Recommended practice to abbreviate names and to prefix same or similar attribute from multiple tables.
  ```

Aliasing, Renaming, and Tuple Variables (2/2)

- The attribute names can also be renamed

```
EMPLOYEE AS E(Fn, Mi, Ln, Ssn, Bd,
  Addr, Sex, Sal, Sssn, Dno)
```

- Note that the relation EMPLOYEE now has a variable name E which corresponds to a tuple variable
- The “AS” may be dropped in most SQL implementations

Explicit Sets and Renaming of Attributes in SQL

- Can use explicit set of values in WHERE clause
- Use qualifier AS followed by desired new name
  - Rename any attribute that appears in the result of a query

```
Q8A:   SELECT   E.Lname AS Employee_name, S.Lname AS Supervisor_name
       FROM     EMPLOYEE AS E, EMPLOYEE AS S
       WHERE    E.Super_ssn=S.Ssn;
```

Unspecified WHERE Claude and Use of the Asterisk (1/2)

- Missing `WHERE` clause
  - Indicates no condition on tuple selection
- Effect is a `CROSS PRODUCT`
  - Result is all possible tuple combinations (or the Algebra operation of Cartesian Product)

Queries 9 and 10. Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

```
Q9:   SELECT   Ssn
      FROM     EMPLOYEE;

Q10:  SELECT   Ssn, Dname
      FROM     EMPLOYEE, DEPARTMENT;
```

Unspecified WHERE Clause and Use of the Asterisk (2/2)

- Specify an asterisk (*)
  - Retrieve all the attribute values of the selected tuples
  - The * can be prefixed by the relation name; e.g., EMPLOYEE *

```
Q1C:    SELECT    *
        FROM      EMPLOYEE
        WHERE     Dno=5;

Q1D:    SELECT    *
        FROM      EMPLOYEE, DEPARTMENT
        WHERE     Dname='Research' AND Dno=Dnumber;

Q10A:   SELECT    *
        FROM      EMPLOYEE, DEPARTMENT;
```

Tables as Sets in SQL (1/2)

- SQL does not automatically eliminate duplicate tuples in query results
- For aggregate operations duplicates must be accounted for
- Use the keyword **`DISTINCT`** in the `SELECT` clause
  - Only distinct tuples should remain in the result

Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:   SELECT    ALL Salary
       FROM      EMPLOYEE;

Q11A:  SELECT    DISTINCT Salary
       FROM      EMPLOYEE;
```

Tables as Sets in SQL (2/2)

- Set operations
  - UNION, **EXCEPT** (difference), **INTERSECT**
  - Corresponding multiset operations: UNION ALL, EXCEPT ALL, INTERSECT ALL)
  - Type compatibility is needed for these operations to be valid

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

```
Q4A: (SELECT   DISTINCT Pnumber
      FROM     PROJECT, DEPARTMENT, EMPLOYEE
      WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn
               AND Lname='Smith' )
      UNION
     ( SELECT  DISTINCT Pnumber
      FROM     PROJECT, WORKS_ON, EMPLOYEE
      WHERE    Pnumber=Pno AND Essn=Ssn
               AND Lname='Smith' );
```

Substring Pattern Matching and Arithmetic Operators

- **`LIKE`** comparison operator
  - Used for string **pattern matching**
  - % replaces an arbitrary number of zero or more characters
  - underscore (_) replaces a single character
  - Examples: **WHERE** Address **LIKE** '%Houston,TX%';
  - **WHERE** Ssn **LIKE** '_ _ 1_ _ 8901';
- **`BETWEEN`** comparison operator

E.g., **WHERE**(Salary **BETWEEN** 30000 **AND** 40000)
AND Dno = 5;

Arithmetic Operations

- Standard arithmetic operators:
  - Addition (+), subtraction (–), multiplication (*), and division (/) may be included as a part of **SELECT**

- **Query 13.** Show the resulting salaries if every employee working on the ‘ProductX’ project is given a 10 percent raise.

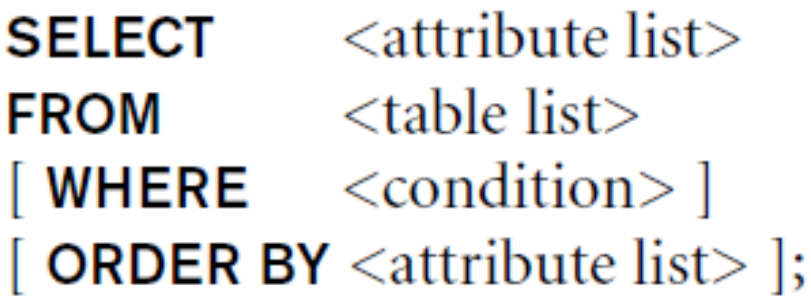
```
SELECT  E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
FROM  EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE  E.Ssn=W.Essn AND W.Pno=P.Pnumber AND
    P.Pname=‘ProductX’;
```

Ordering of Query Results

- Use **`ORDER BY`** clause
  - Keyword **`DESC`** to see result in a descending order of values
  - Keyword **`ASC`** to specify ascending order explicitly
  - Typically placed at the end of the query

```
ORDER BY D.Dname DESC, E.Lname ASC,
  E.Fname ASC
```

Basic SQL Retrieval Query Block



INSERT, DELETE, and UPDATE Statements in SQL

- Three commands used to modify the database:
  - `INSERT`, `DELETE`, and `UPDATE`
- `INSERT` typically inserts a tuple (row) in a relation (table)
- `UPDATE` may update a number of tuples (rows) in a relation (table) that satisfy the condition
- `DELETE` may also update a number of tuples (rows) in a relation (table) that satisfy the condition

INSERT

- 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
- Constraints on data types are observed automatically
- Any integrity constraints as a part of the DDL specification are enforced

The INSERT Command

- Specify the relation name and a list of values for the tuple

```
U1:   INSERT INTO   EMPLOYEE
      VALUES        ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                      Oak Forest, Katy, TX', 'M', 37000, '653298653', 4 );
```

- The variation below inserts multiple tuples where a new table is loaded values from the result of a query.

```
U3B:  INSERT INTO   WORKS_ON_INFO ( Emp_name, Proj_name,
                    Hours_per_week )
      SELECT        E.Lname, P.Pname, W.Hours
      FROM          PROJECT P, WORKS_ON W, EMPLOYEE E
      WHERE         P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

Bulk Loading of Tables

- Another variation of **INSERT** is used for bulk-loading of several tuples into tables
- A new table TNEW can be created with the same attributes as T and using LIKE and DATA in the syntax, it can be loaded with entire data.
- EXAMPLE:

```
CREATE TABLE D5EMPS  LIKE  EMPLOYEE
        (SELECT   E.*
         FROM      EMPLOYEE AS E
        WHERE     E.Dno=5)
 WITH DATA;
```

DELETE

- Removes tuples from a relation
  - Includes a WHERE-clause to select the tuples to be deleted
  - Referential integrity should be enforced
  - Tuples are deleted from only *one table* at a time (unless CASCADE is specified on a referential integrity constraint)
  - 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

The DELETE Command

- Removes tuples from a relation
  - Includes a `WHERE` clause to select the tuples to be deleted. The number of tuples deleted will vary.

```
U4A:  DELETE FROM   EMPLOYEE
      WHERE         Lname='Brown';

U4B:  DELETE FROM   EMPLOYEE
      WHERE         Ssn='123456789';

U4C:  DELETE FROM   EMPLOYEE
      WHERE         Dno=5;

U4D:  DELETE FROM   EMPLOYEE;
```

UPDATE (1/3)

- 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 specified as part of DDL specification is enforced

UPDATE (2/3)

- 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 (3/3)

- 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

Additional Features of SQL (1/2)

- Techniques for specifying complex retrieval queries
- Writing programs in various programming languages that include SQL statements: Embedded and dynamic SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC, SQL/PSM (Persistent Stored Module)
- Set of commands for specifying physical database design parameters, file structures for relations, and access paths, e.g., CREATE INDEX

Additional Features of SQL (2/2)

- Transaction control commands
- Specifying the granting and revoking of privileges to users
- Constructs for creating triggers
- Enhanced relational systems known as object-relational define relations as classes. Abstract data types (called User Defined Types- UDTs) are supported with CREATE TYPE
- New technologies such as XML and OLAP are added to versions of SQL

Summary (1/2)

- SQL
  - A Comprehensive language for relational database management
  - Data definition, queries, updates, constraint specification, and view definition
- Covered :
  - Data definition commands for creating tables
  - Commands for constraint specification
  - Simple retrieval queries
  - Database update commands

Summary (2/2)

- Topics Covered in Next Section
  - More Complex SQL Retrieval Queries
  - Specifying Semantic Constraints as Assertions and Actions as Triggers
  - Views (Virtual Tables) in SQL
  - Schema Modification in SQL

Agenda

1. Session Overview
2. Basic SQL
3. Advanced SQL
4. Introduction to SQL Programming
5. Web Database Programming Using PhP
6. Summary and Conclusion

Agenda

- Complex SQL:
  - Nested queries, joined tables (in the FROM clause), outer joins, aggregate functions, grouping
- Handling semantic constraints with `CREATE ASSERTION` and `CREATE TRIGGER`
- `CREATE VIEW` statement and materialization strategies
- Schema Modification for the DBAs using `ALTER TABLE , ADD and DROP COLUMN, ALTER CONSTRAINT` etc.

More Complex SQL Retrieval Queries

- Additional features allow users to specify more complex retrievals from database:
  - Nested queries, joined tables, and outer joins (in the FROM clause), aggregate functions, and grouping

Comparisons Involving NULLand Three-Valued Logic (1/3)

- Meanings of `NULL`
  - **Unknown value**
  - **Unavailable or withheld value**
  - **Not applicable attribute**
- Each individual `NULL` value considered to be different from every other `NULL` value
- SQL uses a three-valued logic:
  - `TRUE`, `FALSE`, and `UNKNOWN` (like Maybe)
- **NULL = NULL comparison is avoided**

Comparisons Involving NULL and Three-Valued Logic (2/3)

Table 7.1 Logical Connectives in Three-Valued Logic

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Comparisons Involving NULL and Three-Valued Logic (3/3)

- SQL allows queries that check whether an attribute value is `NULL`
  - `IS` or `IS NOT NULL`

Query 18. Retrieve the names of all employees who do not have supervisors.

```
Q18:  SELECT  Fname, Lname
      FROM    EMPLOYEE
      WHERE   Super_ssn IS NULL;
```

Nested Queries, Tuples, and Set/Multiset Comparisons (1/5)

- **Nested queries**
  - Complete select-from-where blocks within WHERE clause of another query
  - **Outer query and nested subqueries**
- Comparison operator `IN`
  - Compares value *v* with a set (or multiset) of values *V*
  - Evaluates to `TRUE` if *v* is one of the elements in *V*

Nested Queries (2/5)

```
Q4A:  SELECT   DISTINCT Pnumber
      FROM     PROJECT
      WHERE    Pnumber IN
               ( SELECT     Pnumber
                 FROM       PROJECT, DEPARTMENT, EMPLOYEE
                 WHERE      Dnum=Dnumber AND
                            Mgr_ssn=Ssn AND Lname='Smith' )
               OR
               Pnumber IN
               ( SELECT     Pno
                 FROM       WORKS_ON, EMPLOYEE
                 WHERE      Essn=Ssn AND Lname='Smith' );
```

Nested Queries (3/5)

- Use tuples of values in comparisons
  - » Place them within parentheses

```
SELECT    DISTINCT Essn
FROM      WORKS_ON
WHERE     (Pno, Hours) IN ( SELECT    Pno, Hours
                            FROM      WORKS_ON
                            WHERE     Essn='123456789' );
```

Nested Queries (4/5)

- Use other comparison operators to compare a single value *v*
  - = ANY (or = SOME) operator
    - Returns TRUE if the value *v* is equal to some value in the set *V* and is hence equivalent to IN
  - Other operators that can be combined with ANY (or SOME): >, >=, <, <=, and <>
  - ALL: value must exceed all values from nested query

```
SELECT   Lname, Fname
FROM     EMPLOYEE
WHERE    Salary > ALL ( SELECT   Salary
                        FROM     EMPLOYEE
                        WHERE    Dno=5 );
```

Nested Queries (5/5)

- Avoid potential errors and ambiguities
  - » Create tuple variables (aliases) for all tables referenced in SQL query

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:  SELECT   E.Fname, E.Lname
      FROM     EMPLOYEE AS E
      WHERE    E.Ssn IN ( SELECT   Essn
                          FROM     DEPENDENT AS D
                          WHERE    E.Fname=D.Dependent_name
                          AND E.Sex=D.Sex );
```

Correlated Nested Queries

- **Queries that are nested using the = or IN comparison operator** can be collapsed into one single block: E.g., Q16 can be written as:

- **Q16A:**
```
SELECT   E.Fname, E.Lname
FROM     EMPLOYEE AS E, DEPENDENT AS D
WHERE    E.Ssn=D.Essn AND E.Sex=D.Sex
                      AND
         E.Fname=D.Dependent_name;
```

- **Correlated** nested query
  - Evaluated once for each tuple in the outer query

The EXISTS and UNIQUE Functions in SQL for correlating queries

- `EXISTS` function
  - Check whether the result of a correlated nested query is empty or not. They are Boolean functions that return a TRUE or FALSE result.
- `EXISTS` and `NOT EXISTS`
  - Typically used in conjunction with a correlated nested query
- SQL function `UNIQUE(Q)`
  - Returns `TRUE` if there are no duplicate tuples in the result of query Q

USE of EXISTS

Q7:

```
SELECT Fname, Lname
    FROM Employee
WHERE EXISTS (SELECT *
                FROM DEPENDENT
                WHERE Ssn= Essn)

          AND EXISTS (SELECT   *
                        FROM Department
                       WHERE Ssn= Mgr_Ssn)
```

USE OF NOT EXISTS

To achieve the “for all” (universal quantifier) effect, we use double negation this way in SQL:

Query: List first and last name of employees who work on ALL projects controlled by Dno=5.

```
SELECT Fname, Lname
    FROM Employee
WHERE NOT EXISTS ( (SELECT  Pnumber
                    FROM PROJECT
                    WHERE Dno=5)

                EXCEPT (SELECT   Pno
                        FROM WORKS_ON
                        WHERE Ssn= ESsn)
```

The above is equivalent to double negation: List names of those employees for whom there does NOT exist a project managed by department no. 5 that they do NOT work on.

Double Negation to accomplish “for all” in SQL

- **Q3B:** **SELECT** Lname, Fname
  FROM EMPLOYEE
  WHERE **NOT EXISTS** (**SELECT** *
  FROM WORKS_ON B
  WHERE (B.Pno **IN** (**SELECT** Pnumber
  FROM PROJECT
  WHERE Dnum=5
  AND
  NOT EXISTS (**SELECT** *
  FROM WORKS_ON C
  WHERE C.Essn=Ssn
  AND C.Pno=B.Pno)));

The above is a direct rendering of: List names of those employees for whom there does NOT exist a project managed by department no. 5 that they do NOT work on.

Explicit Sets and Renaming of Attributes in SQL

- Can use explicit set of values in WHERE clause

```
Q17:    SELECT    DISTINCT Essn
        FROM      WORKS_ON
        WHERE     Pno IN (1, 2, 3);
```

- Use qualifier AS followed by desired new name
  - Rename any attribute that appears in the result of a query

```
Q8A:    SELECT    E.Lname AS Employee_name, S.Lname AS Supervisor_name
        FROM      EMPLOYEE AS E, EMPLOYEE AS S
        WHERE     E.Super_ssn=S.Ssn;
```

Specifying Joined Tables in the FROM Clause of SQL

- **Joined table**
  - Permits users to specify a table resulting from a join operation in the FROM clause of a query
- The FROM clause in Q1A
  - Contains a single joined table. JOIN may also be called INNER JOIN

```
Q1A:    SELECT    Fname, Lname, Address
        FROM      (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
        WHERE     Dname='Research';
```

Different Types of JOINed Tables in SQL

- Specify different types of join
  - NATURAL JOIN
  - Various types of OUTER JOIN (LEFT, RIGHT, FULL)
- NATURAL JOIN on two relations R and S
  - No join condition specified
  - Is equivalent to an implicit EQUIJOIN condition for each pair of attributes with same name from R and S

NATURAL JOIN

- Rename attributes of one relation so it can be joined with another using NATURAL JOIN:

```
Q1B:    SELECT    Fname, Lname, Address
        FROM      (EMPLOYEE NATURAL JOIN
                  (DEPARTMENT AS DEPT (Dname, Dno, Mssn,
                              Msdate)))
        WHERE     Dname='Research';
```

The above works with EMPLOYEE.Dno = DEPT.Dno as an implicit join condition

INNER and OUTER Joins

- INNER JOIN **(versus** OUTER JOIN**)**
  - Default type of join in a joined table
  - Tuple is included in the result only if a matching tuple exists in the other relation
- LEFT OUTER JOIN
  - Every tuple in left table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of right table
- RIGHT OUTER JOIN
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of left table

Example: LEFT OUTER JOIN

SELECT E.Lname **AS** Employee_Name
S.Lname **AS** Supervisor_Name

FROM Employee **AS** E **LEFT OUTER JOIN** EMPLOYEE **AS** S
ON E.Super_ssn = S.Ssn)

ALTERNATE SYNTAX:

SELECT E.Lname **,** S.Lname
FROM EMPLOYEE E, EMPLOYEE S
WHERE E.Super_ssn + = S.Ssn

Multiway JOIN in the FROM clause

- FULL OUTER JOIN – combines result if LEFT and RIGHT OUTER JOIN
- Can nest JOIN specifications for a multiway join:

```
Q2A:    SELECT Pnumber, Dnum, Lname, Address, Bdate
        FROM   ((PROJECT JOIN DEPARTMENT ON
                        Dnum=Dnumber)  JOIN EMPLOYEE ON
                        Mgr_ssn=Ssn)
        WHERE  Plocation='Stafford';
```

Aggregate Functions in SQL (1/2)

- Used to summarize information from multiple tuples into a single-tuple summary
- Built-in aggregate functions
  - **`COUNT`**, **`SUM`**, **`MAX`**, **`MIN`**, and **`AVG`**
- **Grouping**
  - Create subgroups of tuples before summarizing
- To select entire groups, `HAVING` clause is used
- Aggregate functions can be used in the `SELECT` clause or in a `HAVING` clause

Renaming Results of Aggregation

- Following query returns a single row of computed values from EMPLOYEE table:

```
Q19:    SELECT   SUM (Salary), MAX (Salary), MIN (Salary),
        AVG                    (Salary)
            FROM  EMPLOYEE;
```

- The result can be presented with new names:

```
Q19A:   SELECT   SUM (Salary) AS Total_Sal, MAX (Salary) AS
                 Highest_Sal, MIN (Salary) AS Lowest_Sal,
        AVG                    (Salary) AS Average_Sal
            FROM  EMPLOYEE;
```

Aggregate Functions in SQL (2/2)

- NULL values are discarded when aggregate functions are applied to a particular column

Query 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:  SELECT  SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
      FROM    (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
      WHERE   Dname='Research';
```

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21:  SELECT  COUNT (*)
      FROM    EMPLOYEE;

Q22:  SELECT  COUNT (*)
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DNO=DNUMBER AND DNAME='Research';
```

Aggregate Functions on Booleans

- SOME and ALL may be applied as functions on Boolean Values.
- SOME returns true if at least one element in the collection is TRUE (similar to OR)
- ALL returns true if all of the elements in the collection are TRUE (similar to AND)

Grouping: The GROUP BY Clause (1/2)

- **Partition** relation into subsets of tuples
  - Based on **grouping attribute(s)**
  - Apply function to each such group independently
- **`GROUP BY`** clause
  - Specifies grouping attributes
- COUNT (*) counts the number of rows in the group

Examples of GROUP BY

- The grouping attribute must appear in the SELECT clause:

```
Q24:    SELECT    Dno, COUNT (*), AVG (Salary)
        FROM      EMPLOYEE
        GROUP BY  Dno;
```

- If the grouping attribute has NULL as a possible value, then a separate group is created for the null value (e.g., null Dno in the above query)
- GROUP BY may be applied to the result of a JOIN:

```
Q25:    SELECT    Pnumber, Pname, COUNT (*)
        FROM      PROJECT, WORKS_ON
        WHERE     Pnumber=Pno
        GROUP BY  Pnumber, Pname;
```

Grouping: The GROUP BY and HAVING Clauses (2/2)

- **`HAVING`** clause
  - Provides a condition to select or reject an entire group:
- **Query 26.** For each project *on which more than two employees work,* retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26:    SELECT      Pnumber, Pname, COUNT (*)
        FROM        PROJECT, WORKS_ON
        WHERE       Pnumber=Pno
        GROUP BY    Pnumber, Pname
        HAVING      COUNT (*) > 2;
```

Combining the WHERE and the HAVING Clause (1/2)

- Consider the query: we want to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work.

- INCORRECT QUERY:

```
SELECT      Dno, COUNT (*)
FROM        EMPLOYEE
WHERE       Salary>40000
GROUP BY    Dno
HAVING      COUNT (*) > 5;
```

Correct Specification of the Query:

- Note: the WHERE clause applies tuple by tuple whereas HAVING applies to entire group of tuples

Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28:  SELECT   Dnumber, COUNT (*)
      FROM     DEPARTMENT, EMPLOYEE
      WHERE    Dnumber=Dno AND Salary>40000 AND
               ( SELECT       Dno
                 FROM         EMPLOYEE
                 GROUP BY Dno
                 HAVING       COUNT (*) > 5)
```

Use of WITH

- The WITH clause allows a user to define a table that will only be used in a particular query (not available in all SQL implementations)
- Used for convenience to create a temporary “View” and use that immediately in a query
- Allows a more straightforward way of looking a step-by-step query

Example of WITH

- See an alternate approach to doing Q28:

- **Q28':**

```
WITH    BIGDEPTS (Dno) AS
( SELECT  Dno
  FROM    EMPLOYEE
  GROUP BY Dno
  HAVING  COUNT (*) > 5)
SELECT          Dno, COUNT (*)
FROM            EMPLOYEE
WHERE           Salary>40000 AND Dno IN BIGDEPTS
GROUP BY Dno;
```

Use of CASE

- SQL also has a CASE construct
- Used when a value can be different based on certain conditions.
- Can be used in any part of an SQL query where a value is expected
- Applicable when querying, inserting or updating tuples

EXAMPLE of use of CASE

- The following example shows that employees are receiving different raises in different departments (A variation of the update U6)

- **U6':** **UPDATE** EMPLOYEE
  SET Salary =
  CASE **WHEN** Dno = 5 **THEN** Salary + 2000
  WHEN Dno = 4 **THEN** Salary + 1500
  WHEN Dno = 1 **THEN** Salary + 3000

Recursive Queries in SQL

- An example of a **recursive relationship** between tuples of the same type is the relationship between an employee and a supervisor.
- This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation
- An example of a **recursive operation** is to retrieve all supervisees of a supervisory employee *e* at all levels—that is, all employees *e′* directly supervised by *e*, all employees *e′′* directly supervised by each employee *e′*, all employees *e′′′* directly supervised by each employee *e″*, and so on. Thus the CEO would have each employee in the company as a supervisee in the resulting table. Example shows such table SUP_EMP with 2 columns (Supervisor,Supervisee(any level)):

An EXAMPLE of RECURSIVE Query

- **Q29: WITH RECURSIVE** SUP_EMP (SupSsn, EmpSsn) **AS**

```
        SELECT      SupervisorSsn, Ssn
        FROM  EMPLOYEE
              UNION
        SELECT      E.Ssn, S.SupSsn
        FROM  EMPLOYEE AS E, SUP_EMP AS S
        WHERE       E.SupervisorSsn = S.EmpSsn)
        SELECT            *
        FROM  SUP_EMP;
```

- The above query starts with an empty SUP_EMP and successively builds SUP_EMP table by computing immediate supervisees first, then second level supervisees, etc. until a **fixed point** is reached and no more supervisees can be added

EXPANDED Block Structure of SQL Queries

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

Specifying Constraints as Assertions and Actions as Triggers

- Semantic Constraints: The following are beyond the scope of the EER and relational model
- **`CREATE ASSERTION`**
  - Specify additional types of constraints outside scope of built-in relational model constraints
- **`CREATE TRIGGER`**
  - Specify automatic actions that database system will perform when certain events and conditions occur

Specifying General Constraints as Assertions in SQL

- **CREATE ASSERTION**
  - Specify a query that selects any tuples that violate the desired condition
  - Use only in cases where it goes beyond a simple `CHECK` which applies to individual attributes and domains

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS  ( SELECT     *
                      FROM       EMPLOYEE E, EMPLOYEE M,
                                 DEPARTMENT D
                      WHERE      E.Salary>M.Salary
                                 AND E.Dno=D.Dnumber
                                 AND D.Mgr_ssn=M.Ssn ) );
```

Introduction to Triggers in SQL

- `CREATE TRIGGER` statement
  - Used to monitor the database
- Typical trigger has three components which make it a rule for an “active database “ (more on active databases in section 26.1) :
  - **Event(s)**
  - **Condition**
  - **Action**

USE OF TRIGGERS

- AN EXAMPLE with standard Syntax.(Note : other SQL implementations like PostgreSQL use a different syntax.)

R5:

CREATE TRIGGER SALARY_VIOLATION
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)

Views (Virtual Tables) in SQL

- Concept of a view in SQL
  - Single table derived from other tables called the **defining tables**
  - Considered to be a virtual table that is not necessarily populated

Specification of Views in SQL (1/2)

- **`CREATE VIEW`** command
  - Give table name, list of attribute names, and a query to specify the contents of the view
  - In V1, attributes retain the names from base tables. In V2, attributes are assigned names

```
V1:   CREATE VIEW   WORKS_ON1
      AS SELECT     Fname, Lname, Pname, Hours
          FROM      EMPLOYEE, PROJECT, WORKS_ON
          WHERE     Ssn=Essn AND Pno=Pnumber;

V2:   CREATE VIEW   DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT     Dname, COUNT (*), SUM (Salary)
          FROM      DEPARTMENT, EMPLOYEE
          WHERE     Dnumber=Dno
          GROUP BY  Dname;
```

Specification of Views in SQL (2/2)

- Once a View is defined, SQL queries can use the View relation in the FROM clause
- View is always up-to-date
  - » Responsibility of the DBMS and not the user
- **`DROP VIEW`** command
  - » Dispose of a view

View Implementation, View Update, and Inline Views

- Complex problem of efficiently implementing a view for querying
- **Strategy1: Query modification** approach
  - Compute the view as and when needed. Do not store permanently
  - Modify view query into a query on underlying base tables
  - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute

View Materialization (1/2)

- **Strategy 2: View materialization**
  - Physically create a temporary view table when the view is first queried
  - Keep that table on the assumption that other queries on the view will follow
  - Requires efficient strategy for automatically updating the view table when the base tables are updated
- **Incremental update strategy for materialized views**
  - DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table

View Materialization (2/2)

- Multiple ways to handle materialization:
  - **immediate update** strategy updates a view as soon as the base tables are changed
  - **lazy update** strategy updates the view when needed by a view query
  - **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date). This is commonly used in Banks, Retail store operations, etc.

View Update

- Update on a view defined on a single table without any aggregate functions
  - » Can be mapped to an update on underlying base table- possible if the primary key is preserved in the view
- Update not permitted on aggregate views. E.g.,

```
UV2:    UPDATE      DEPT_INFO
        SET         Total_sal=100000
        WHERE       Dname='Research';
```

cannot be processed because Total_sal is a computed value in the view definition

View Update and Inline Views

- View involving joins
  - Often not possible for DBMS to determine which of the updates is intended
- Clause **`WITH CHECK OPTION`**
  - Must be added at the end of the view definition if a view is to be updated to make sure that tuples being updated stay in the view
- **In-line view**
  - Defined in the `FROM` clause of an SQL query (e.g., we saw its used in the WITH example)

Views as authorization mechanism

- SQL query authorization statements (GRANT and REVOKE) are described in detail in Chapter 30
- Views can be used to hide certain attributes or tuples from unauthorized users
- E.g., For a user who is only allowed to see employee information for those who work for department 5, he may only access the view DEPT5EMP:

```
CREATE VIEW   DEPT5EMP  AS
SELECT        *
FROM          EMPLOYEE
WHERE         Dno = 5;
```

Schema Change Statements in SQL

- **Schema evolution commands**
  - DBA may want to change the schema while the database is operational
  - Does not require recompilation of the database schema

The DROP Command

- `DROP` command
  - Used to drop named schema elements, such as tables, domains, or constraint
- Drop behavior options:
  - `CASCADE` and `RESTRICT`
- Example:
  - `DROP SCHEMA COMPANY CASCADE;`
  - This removes the schema and all its elements including tables,views, constraints, etc.

The ALTER table command

- **Alter table actions** include:
  - Adding or dropping a column (attribute)
  - Changing a column definition
  - Adding or dropping table constraints
- Example:
  - `ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);`

Adding and Dropping Constraints

- Change constraints specified on a table
  - Add or drop a named constraint

```
ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK CASCADE;
```

Dropping Columns, Default Values

- To drop a column
  - Choose either `CASCADE` or `RESTRICT`
  - `CASCADE` would drop the column from views etc. `RESTRICT` is possible if no views refer to it.

```
ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN
    Address CASCADE;
```

- Default values can be dropped and altered :

```
ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn
    DROP DEFAULT;
ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn
    SET DEFAULT '333445555';
```

Table 7.2 Summary of SQL Syntax

Table 7.2 Summary of SQL Syntax

```
CREATE TABLE <table name> ( <column name> <column type> [ <attribute constraint> ]
                                { , <column name> <column type> [ <attribute constraint> ] }
                                [ <table constraint> { , <table constraint> } ] )
```

```
DROP TABLE <table name>
ALTER TABLE <table name> ADD <column name> <column type>
```

```
SELECT [ DISTINCT ] <attribute list>
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]
```

```
<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )
                          { , ( <column name> | <function> ( ( [ DISTINCT] <column name> | * ) ) } ) )
```

```
<grouping attributes> ::= <column name> { , <column name> }
```

```
<order> ::= ( ASC | DESC )
```

```
INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) }
| <select statement> )
```

continued on next slide

Table 7.2 (continued) Summary of SQL Syntax

Table 7.2 Summary of SQL Syntax

DELETE FROM <table name> [WHERE <selection condition>]
UPDATE <table name> SET <column name> = <value expression> { , <column name> = <value expression> } [WHERE <selection condition>]
CREATE [UNIQUE] INDEX <index name> ON <table name> (<column name> [<order>] { , <column name> [<order>] }) [CLUSTER]
DROP INDEX <index name>
CREATE VIEW <view name> [(<column name> { , <column name> })] AS <select statement>
DROP VIEW <view name>

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

Summary (1/2)

- Complex SQL:
  - » Nested queries, joined tables (in the FROM clause), outer joins, aggregate functions, grouping
- Handling semantic constraints with `CREATE ASSERTION` and `CREATE TRIGGER`
- `CREATE VIEW` statement and materialization strategies
- Schema Modification for the DBAs using `ALTER TABLE , ADD and DROP COLUMN, ALTER CONSTRAINT` etc.

Summary (2/2)

- Topics Covered in Next Section
  - Database Programming: Techniques and Issues
  - Embedded SQL, Dynamic SQL, and SQLJ
  - Database Programming with Function Calls: SQL/CLI and JDBC
  - Database Stored Procedures and SQL/PSM
  - Comparing the Three Approaches

Agenda

1. Session Overview
2. Basic SQL
3. Advanced SQL
4. Introduction to SQL Programming
5. Web Database Programming Using PhP
6. Summary and Conclusion

Agenda

- Database Programming: Techniques and Issues
- Embedded SQL, Dynamic SQL, and SQLJ
- Database Programming with Function Calls: SQL/CLI and JDBC
- Database Stored Procedures and SQL/PSM
- Comparing the Three Approaches

- **Database applications**
  - Host language
    - Java, C/C++/C#, COBOL, or some other programming language
  - Data sublanguage
    - SQL
- SQL standards
  - Continually evolving
  - Each DBMS vendor may have some variations from standard

- **Interactive interface**
  - SQL commands typed directly into a monitor
- Execute **file of commands**
  - @<*filename*>
- **Application programs** or **database applications**
  - Used as canned transactions by the end users access a database
  - May have **Web interface**

Approaches to Database Programming (1/2)

- **Embedding** database commands in a general-purpose programming language
  - Database statements identified by a special prefix
  - **Precompiler** or **preprocessor** scans the source program code
    - Identify database statements and extract them for processing by the DBMS
  - Called **embedded SQL**

Approaches to Database Programming (2/2)

- Using a library of database functions
  - **Library of functions** available to the host programming language
  - **Application programming interface (API)**
- Designing a brand-new language
  - **Database programming language** designed from scratch
- First two approaches are more common

Impedance Mismatch

- Differences between database model and programming language model
- **Binding** for each host programming language
  - Specifies for each attribute type the compatible programming language types
- Cursor or iterator variable
  - Loop over the tuples in a query result

Typical Sequence of Interaction in Database Programming

- Open a connection to database server
- Interact with database by submitting queries, updates, and other database commands
- Terminate or close connection to database

Embedded SQL, Dynamic SQL, and SQLJ

- **Embedded SQL**
  - C language
- **SQLJ**
  - Java language
- Programming language called **host language**

Retrieving Single Tuples with Embedded SQL (1/4)

- `EXEC SQL`
  - Prefix
  - **Preprocessor** separates embedded SQL statements from host language code
  - Terminated by a matching `END-EXEC`
    - Or by a semicolon (;)
- **Shared variables**
  - Used in both the C program and the embedded SQL statements
  - Prefixed by a colon (:) in SQL statement

Figure 10.1 C program variables used in the embedded SQL ex. E1 & E2

```
0)  int loop ;
1)  EXEC SQL BEGIN DECLARE SECTION ;
2)  varchar dname [16], fname [16], lname [16], address [31] ;
3)  char ssn [10], bdate [11], sex [2], minit [2] ;
4)  float salary, raise ;
5)  int dno, dnumber ;
6)  int SQLCODE ; char SQLSTATE [6] ;
7)  EXEC SQL END DECLARE SECTION ;
```

Retrieving Single Tuples with Embedded SQL (2/4)

- Connecting to the database

```
CONNECT TO <server name>AS <connection name>
AUTHORIZATION <user account name and password> ;
```

- Change connection

```
SET CONNECTION <connection name> ;
```

- Terminate connection

```
DISCONNECT <connection name> ;
```

Retrieving Single Tuples with Embedded SQL (3/4)

- **SQLCODE** and **SQLSTATE** communication variables
  - Used by DBMS to communicate exception or error conditions
- SQLCODE variable
  - 0 = statement executed successfully
  - 100 = no more data available in query result
  - < 0 = indicates some error has occurred

Retrieving Single Tuples with Embedded SQL (4/4)

- SQLSTATE
  - String of five characters
  - ‘00000’ = no error or exception
  - Other values indicate various errors or exceptions
  - For example, ‘02000’ indicates ‘no more data’ when using SQLSTATE

Figure 10.2 Program segment E1, a C pgm segment with embedded SQL

```
   //Program Segment E1:
0) loop = 1 ;
1) while (loop) {
2)   prompt("Enter a Social Security Number: ", ssn) ;
3)   EXEC SQL
4)     SELECT Fname, Minit, Lname, Address, Salary
5)     INTO :fname, :minit, :lname, :address, :salary
6)     FROM EMPLOYEE WHERE Ssn = :ssn ;
7)   if (SQLCODE = = 0) printf(fname, minit, lname, address, salary)
8)     else printf("Social Security Number does not exist: ", ssn) ;
9)   prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10)  }
```

Retrieving Multiple Tuples with Embedded SQL Using Cursors (1/2)

- Cursor
  - Points to a single tuple (row) from result of query
- **`OPEN CURSOR`** command
  - Fetches query result and sets cursor to a position before first row in result
  - Becomes current row for cursor
- **`FETCH`** commands
  - Moves cursor to next row in result of query

Figure 10.3 Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes

```
   //Program Segment E2:
 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 ;
```

Retrieving Multiple Tuples with Embedded SQL Using Cursors (2/2)

- **`FOR UPDATE OF`**
  - List the names of any attributes that will be updated by the program
- Fetch orientation
  - Added using value: `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` *i*, and `RELATIVE` *i*

DECLARE <cursor name> [**INSENSITIVE**] [**SCROLL**] **CURSOR**
[**WITH HOLD**] **FOR** <query specification>
[**ORDER BY** <ordering specification>]
[**FOR READ ONLY** | **FOR UPDATE** [**OF** <attribute list>]] ;

Specifying Queries at Runtime Using Dynamic SQL

- **Dynamic SQL**
  - Execute different SQL queries or updates dynamically at runtime
- Dynamic update
- Dynamic query

Figure 10.4 Program segment E3, a C program segment that uses dynamic SQL for updating a table

```
    //Program Segment E3:
0)  EXEC SQL BEGIN DECLARE SECTION ;
1)  varchar sqlupdatestring [256] ;
2)  EXEC SQL END DECLARE SECTION ;
    ...
3)  prompt("Enter the Update Command: ", sqlupdatestring) ;
4)  EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5)  EXEC SQL EXECUTE sqlcommand ;
    ...
```

SQLJ: Embedding SQL Commands in Java

- Standard adopted by several vendors for embedding SQL in Java
- Import several class libraries
- **Default context**
- Uses **exceptions** for error handling
  - `SQLException` is used to return errors or exception conditions

Figure 10.5 Importing classes needed for including SQLJ in Java programs in Oracle, and establishing a connection and default context

```
1)  import java.sql.* ;
2)  import java.io.* ;
3)  import sqlj.runtime.* ;
4)  import sqlj.runtime.ref.* ;
5)  import oracle.sqlj.runtime.* ;
    ...
6)  DefaultContext cntxt =
7)  oracle.getConnection("<url name>", "<user name>", "<password>", true) ;
8)  DefaultContext.setDefaultContext(cntxt) ;
    ...
```

Figure 10.6 Java program variables used in SQLJ examples J1 and J2

```
1)  string dname, ssn , fname, fn, lname, ln,
    bdate, address ;
2)  char sex, minit, mi ;
3)  double salary, sal ;
4)  integer dno, dnumber ;
```

Figure 10.7 Program segment J1, a Java program segment with SQLJ

```
   //Program Segment J1:
 1) ssn = readEntry("Enter a Social Security Number: ") ;
 2) try {
 3)   #sql { SELECT Fname, Minit, Lname, Address, Salary
 4)     INTO :fname, :minit, :lname, :address, :salary
 5)     FROM EMPLOYEE WHERE Ssn = :ssn} ;
 6) } catch (SQLException se) {
 7)     System.out.println("Social Security Number does not exist: " + ssn) ;
 8)     Return ;
 9)   }
10) System.out.println(fname + " " + minit + " " + lname + " " + address
      + " " + salary)
```

Retrieving Multiple Tuples in SQLJ Using Iterators

- **Iterator**
  - Object associated with a collection (set or multiset) of records in a query result
- **Named iterator**
  - Associated with a query result by listing attribute names and types in query result
- **Positional iterator**
  - Lists only attribute types in query result

Figure 10.8 Program segment J2A, a Java program segment that uses a named iterator to print employee information in a particular department

```
    //Program Segment J2A:
 0) dname = readEntry("Enter the Department Name: ") ;
 1) try {
 2)   #sql { SELECT Dnumber INTO :dnumber
 3)     FROM DEPARTMENT WHERE Dname = :dname} ;
 4) } catch (SQLException se) {
 5)   System.out.println("Department does not exist: " + dname) ;
 6)   Return ;
 7)   }
 8) System.out.printline("Employee information for Department: " + dname) ;
 9) #sql iterator Emp(String ssn, String fname, String minit, String lname,
      double salary) ;
10) Emp e = null ;
11) #sql e = { SELECT ssn, fname, minit, lname, salary
12)   FROM EMPLOYEE WHERE Dno = :dnumber} ;
13) while (e.next()) {
14)   System.out.printline(e.ssn + " " + e.fname + " " + e.minit + " " +
        e.lname  + " " + e.salary) ;
15) } ;
16) e.close() ;
```

Figure 10.9 Program segment J2B, a Java program segment that uses a positional iterator to print employee information in a particular department

```
    //Program Segment J2B:
 0) dname = readEntry("Enter the Department Name: ") ;
 1) try {
 2)   #sql { SELECT Dnumber INTO :dnumber
 3)     FROM DEPARTMENT WHERE Dname = :dname} ;
 4) } catch (SQLException se) {
 5)   System.out.println("Department does not exist: " + dname) ;
 6)   Return ;
 7)   }
 8) System.out.printline("Employee information for Department: " + dname) ;
 9) #sql iterator Emppos(String, String, String, String, double) ;
10) Emppos e = null ;
11) #sql e = { SELECT ssn, fname, minit, lname, salary
12)   FROM EMPLOYEE WHERE Dno = :dnumber} ;
13) #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
14) while (!e.endFetch()) {
15)   System.out.printline(ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16)   #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
17) } ;
18) e.close() ;
```

Database Programming with Function Calls: SQL/CLI & JDBC

- Use of function calls
  - **Dynamic** approach for database programming
- Library of functions
  - Also known as **application programming interface (API)**
  - Used to access database
- **SQL Call Level Interface (SQL/CLI)**
  - Part of SQL standard

SQL/CLI: Using C as the Host Language (1/2)

- **Environment record**
  - Track one or more database connections
  - Set environment information
- **Connection record**
  - Keeps track of information needed for a particular database connection
- **Statement record**
  - Keeps track of the information needed for one SQL statement

SQL/CLI: Using C as the Host Language (2/2)

- **Description record**
  - Keeps track of information about tuples or parameters
- **Handle** to the record
  - C pointer variable makes record accessible to program

Figure 10.10 Program segment CLI1, a C program segment with SQL/CLI

```
   //Program CLI1:
 0) #include sqlcli.h ;
 1) void printSal() {
 2) SQLHSTMT stmt1 ;
 3) SQLHDBC con1 ;
 4) SQLHENV env1 ;
 5) SQLRETURN ret1, ret2, ret3, ret4 ;
 6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
 7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
 8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
      SQL_NTS) else exit ;
 9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Ssn = ?",
      SQL_NTS) ;
11) prompt("Enter a Social Security Number: ", ssn) ;
12) SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15)   SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)   SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)   ret2 = SQLFetch(stmt1) ;
18)   if (!ret2) printf(ssn, lname, salary)
19)     else printf("Social Security Number does not exist: ", ssn) ;
20)   }
21) }
```

Figure 10.11 Program segment CLI2, a C program segment that uses SQL/CLI for a query with a collection of tuples in its result

```
   //Program Segment CLI2:
 0) #include sqlcli.h ;
 1) void printDepartmentEmps() {
 2) SQLHSTMT stmt1 ;
 3) SQLHDBC con1 ;
 4) SQLHENV env1 ;
 5) SQLRETURN ret1, ret2, ret3, ret4 ;
 6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
 7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
 8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
      SQL_NTS) else exit ;
 9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Dno = ?",
      SQL_NTS) ;
11) prompt("Enter the Department Number: ", dno) ;
12) SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15)   SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)   SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)   ret2 = SQLFetch(stmt1) ;
18)   while (!ret2) {
19)     printf(lname, salary) ;
20)     ret2 = SQLFetch(stmt1) ;
21)     }
22)   }
23) }
```

JDBC: SQL Function Calls for Java Programming (1/2)

- JDBC
  - Java function libraries
- Single Java program can connect to several different databases
  - Called data sources accessed by the Java program
- `Class.forName("oracle.jdbc.driver.OracleDriver")`
  - Load a **JDBC driver** explicitly

JDBC: SQL Function Calls for Java Programming (2/2)

- **`Connection` object**
- **`Statement` object** has two subclasses:
  - `PreparedStatement` and `CallableStatement`
- Question mark (`?`) symbol
  - Represents a statement parameter
  - Determined at runtime
- **`ResultSet` object**
  - Holds results of query

Figure 10.12 Program segment JDBC1, a Java program segment with JDBC

```
   //Program JDBC1:
 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) }
```

Figure 10.13 Program segment JDBC2, a Java program segment that uses JDBC for a query with a collection of tuples in its result

```
   //Program Segment JDBC2:
 0) import java.io.* ;
 1) import java.sql.*
   ...
 2) class printDepartmentEmps {
 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, lname ;
 9)     Double salary ;
10)     Integer dno ;
11)     dbacct = readentry("Enter database account:") ;
12)     passwrd = readentry("Enter password:") ;
13)     Connection conn = DriverManager.getConnection
14)       ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
15)     dno = readentry("Enter a Department Number: ") ;
16)     String q = "select Lname, Salary from EMPLOYEE where Dno = " +
        dno.tostring() ;
17)     Statement s = conn.createStatement() ;
18)     ResultSet r = s.executeQuery(q) ;
19)     while (r.next()) {
20)       lname = r.getString(1) ;
21)       salary = r.getDouble(2) ;
22)       system.out.println(lname + salary) ;
23)   } }
24) }
```

Database Stored Procedures and SQL/PSM

- **Stored procedures**
  - Program modules stored by the DBMS at the database server
  - Can be functions or procedures
- SQL/PSM (**SQL/Persistent Stored Modules**)
  - Extensions to SQL
  - Include general-purpose programming constructs in SQL

Database Stored Procedures and Functions (1/2)

- **Persistent stored modules**
  - Stored persistently by the DBMS
- Useful:
  - When database program is needed by several applications
  - To reduce data transfer and communication cost between client and server in certain situations
  - To enhance modeling power provided by views

Database Stored Procedures and Functions (2/2)

- Declaring stored procedures:

```
CREATE PROCEDURE <procedure name> (<parameters>)
<local declarations>
<procedure body> ;
declaring a function, a return type is necessary,
  so the declaration form is
CREATE FUNCTION <function name> (<parameters>)
RETURNS <return type>
<local declarations>
<function body> ;
```

Database Stored Procedures and Functions (3/3)

- Each parameter has parameter type
  - **Parameter type**: one of the SQL data types
  - **Parameter mode**: `IN`, `OUT`, or `INOUT`
- Calling a stored procedure:

```
CALL <procedure or function name>
(<argument list>) ;
```

SQL/PSM: Extending SQL for Specifying Persistent Stored Modules (1/2)

- Conditional branching statement:

```
IF <condition> THEN <statement list>
ELSEIF <condition> THEN <statement list>
...
ELSEIF <condition> THEN <statement list>
ELSE <statement list>
END IF ;
```

SQL/PSM (2/2)

- Constructs for looping

```
WHILE <condition> DO
        <statement list>
END WHILE ;
REPEAT
        <statement list>
UNTIL <condition>
END REPEAT ;

FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
        <statement list>
END FOR ;
```

Figure 10.14 Declaring a function in SQL/PSM

```
//Function PSM1:
0) CREATE FUNCTION Dept_size(IN deptno INTEGER)
1) RETURNS VARCHAR [7]
2) DECLARE No_of_emps INTEGER ;
3) SELECT COUNT(*) INTO No_of_emps
4) FROM EMPLOYEE WHERE Dno = deptno ;
5) IF No_of_emps > 100 THEN RETURN "HUGE"
6) ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
7) ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
8) ELSE RETURN "SMALL"
9) END IF ;
```

Comparing the Three Approaches (1/2)

- Embedded SQL Approach
  - Query text checked for syntax errors and validated against database schema at compile time
  - For complex applications where queries have to be generated at runtime
    - Function call approach more suitable

Comparing the Three Approaches (2/2)

- Library of Function Calls Approach
  - More flexibility
  - More complex programming
  - No checking of syntax done at compile time
- Database Programming Language Approach
  - Does not suffer from the impedance mismatch problem
  - Programmers must learn a new language

Summary (1/2)

- Database Programming: Techniques and Issues
- Embedded SQL, Dynamic SQL, and SQLJ
- Database Programming with Function Calls: SQL/CLI and JDBC
- Database Stored Procedures and SQL/PSM
- Comparing the Three Approaches

Summary (2/2)

- Topics Covered in Next Section
  - A Simple PHP Example
  - Overview of Basic Features of PHP
  - Overview of PHP Database Programming

Agenda

1. Session Overview
2. Basic SQL
3. Advanced SQL
4. Introduction to SQL Programming
5. Web Database Programming Using PhP
6. Summary and Conclusion

Agenda

- A Simple PHP Example
- Overview of Basic Features of PHP
- Overview of PHP Database Programming

Web Database Programming Using PHP

- Techniques for programming dynamic features into Web
- PHP
  - Open source scripting language
  - Interpreters provided free of charge
  - Available on most computer platforms

A Simple PHP Example (1/5)

- PHP
  - Open source general-purpose scripting language
  - Comes installed with the UNIX operating system

A Simple PHP Example (2/5)

- DBMS
  - **Bottom-tier database server**
- PHP
  - **Middle-tier Web server**
- HTML
  - **Client tier**

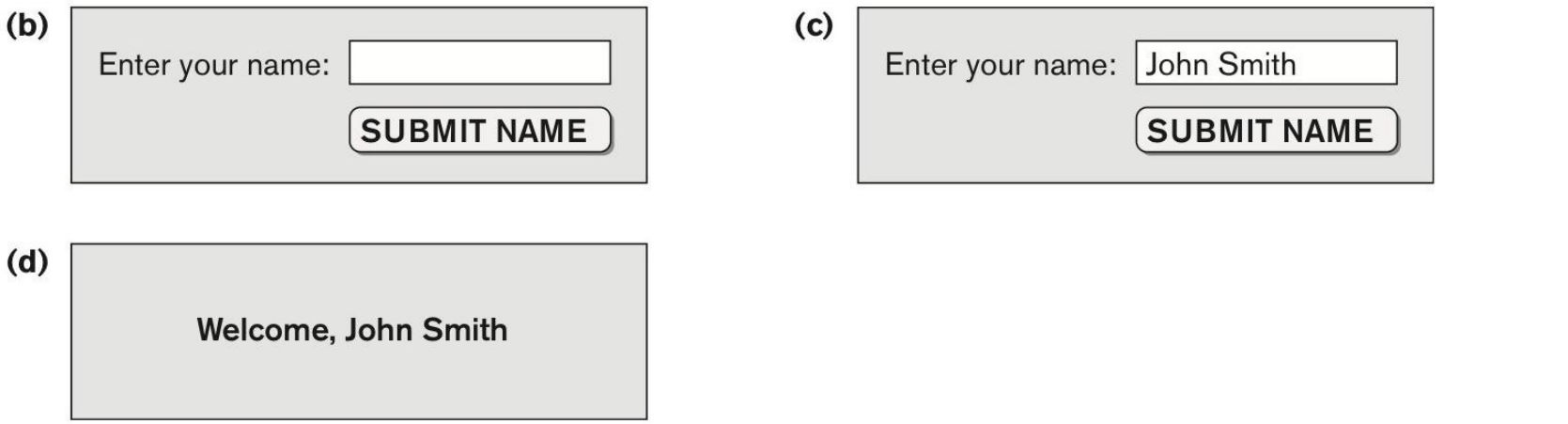
Figure 11.1a PHP program segment for entering a greeting

(a)

```
    //Program Segment P1:
 0) <?php
 1) // Printing a welcome message if the user submitted their name
    // through the HTML form
 2) if ($_POST['user_name']) {
 3)   print("Welcome, ") ;
 4)   print($_POST['user_name']);
 5) }
 6) else {
 7)   // Printing the form to enter the user name since no name has
      // been entered yet
 8)   print <<<_HTML_
 9)   <FORM method="post" action="$_SERVER['PHP_SELF']">
10)   Enter your name: <input type="text" name="user_name">
11)   <BR/>
12)   <INPUT type="submit" value="SUBMIT NAME">
13)   </FORM>
14)   _HTML_;
15) }
16) ?>
```

continued on next slide

Figure 11.1b-d (b) Initial form displayed by PHP program segment. (c) User enters name John Smith. (d) Form prints welcome message for John Smith



A Simple PHP Example (3/5)

- Example Figure 11.1(a)
- PHP script stored in:
  - http://www.myserver.com/example/greeting.php
- `<?php`
  - PHP start tag
- `?>`
  - PHP end tag
- Comments: `//` or `/* */`

A Simple PHP Example (4/5)

- `$_POST`
  - **Auto-global** predefined PHP variable
  - Array that holds all the values entered through form parameters
- Arrays are dynamic
- **Long text strings**
  - Between opening `<<<_HTML_` and closing `_HTML_;`

A Simple PHP Example (5/5)

- PHP **variable names**
  - Start with `$` sign

Overview of Basic Features of PHP

- Illustrate features of PHP suited for creating dynamic Web pages that contain database access commands

PHP Variables, Data Types, and Programming Constructs (1/4)

- PHP **variable names**
  - Start with `$` symbol
  - Can include characters, letters, and underscore character (_)
- Main ways to express strings and text
  - **Single-quoted strings**
  - **Double-quoted strings**
  - **Here documents**
  - **Single and double quotes**

PHP Variables, Data Types, and Programming Constructs (2/4)

- Period (.) symbol
  - String concatenate operator
- Single-quoted strings
  - Literal strings that contain no PHP program variables
- Double-quoted strings and here documents
  - Values from variables need to be interpolated into string

PHP Variables, Data Types, and Programming Constructs (3/4)

- Numeric data types
  - Integers and floating points
- Programming language constructs
  - For-loops, while-loops, and conditional if-statements
- Boolean expressions

Figure 11.2 Illustrating basic PHP string and text values

```
 0) print 'Welcome to my Web site.';
 1) print 'I said to him, "Welcome Home"';
 2) print 'We\'ll now visit the next Web site';
 3) printf('The cost is $%.2f and the tax is $%.2f',
    $cost, $tax) ;
 4) print strtolower('AbCdE');
 5) print ucwords(strtolower('JOHN smith'));
 6) print 'abc' . 'efg'
 7) print "send your email reply to: $email_address"
 8) print <<<FORM_HTML
 9) <FORM method="post" action="$_SERVER['PHP_SELF']">
10) Enter your name: <input type="text" name="user_name">
11) FORM_HTML
```

PHP Variables, Data Types, and Programming Constructs (4/4)

- Comparison operators
  - == (equal), != (not equal), > (greater than), >= (greater than or equal), < (less than), and <= (less than or equal)

PHP Arrays (1/3)

- Can hold database query results
  - Two-dimensional arrays
  - First dimension representing rows of a table
  - Second dimension representing columns (attributes) within a row
- Main types of arrays:
  - **Numeric** and **associative**

PHP Arrays (2/3)

- Numeric array
  - Associates a numeric index with each element in the array
  - Indexes are integer numbers
    - Start at zero
    - Grow incrementally
- Associative array
  - Provides pairs of (key => value) elements

Figure 11.3 Illustrating basic PHP array processing

```
 0) $teaching = array('Database' => 'Smith', 'OS' => 'Carrick',
                      'Graphics' => 'Kam');
 1) $teaching['Graphics'] = 'Benson'; $teaching['Data Mining'] = 'Li';
 2) sort($teaching);
 3) foreach ($teaching as $key => $value) {
 4)   print " $key : $value\n";}
 5) $courses = array('Database', 'OS', 'Graphics', 'Data Mining');
 6) $alt_row_color = array('blue', 'yellow');
 7) for ($i = 0, $num = count($courses); i < $num; $i++) {
 8)   print '<TR bgcolor="' . $alt_row_color[$i % 2] . '">';
 9)   print "<TD>Course $i is</TD><TD>$course[$i]</TD></TR>\n";
10) }
```

PHP Arrays (3/3)

- Techniques for looping through arrays in PHP
- Count function
  - » Returns current number of elements in array
- Sort function
  - » Sorts array based on element values in it

PHP Functions

- Functions
  - Define to structure a complex program and to share common sections of code
  - Arguments passed by value
- Examples to illustrate basic PHP functions
  - Figure 11.4
  - Figure 11.5

Figure 11.4 Rewriting program segment P1 as P1' using functions

```
   //Program Segment P1′:
 0) function display_welcome() {
 1)     print("Welcome, ") ;
 2)     print($_POST['user_name']);
 3) }
 4)
 5) function display_empty_form(); {
 6) print <<<_HTML_
 7) <FORM method="post" action="$_SERVER['PHP_SELF']">
 8) Enter your name: <INPUT type="text" name="user_name">
 9) <BR/>
10) <INPUT type="submit" value="Submit name">
11) </FORM>
12) _HTML_;
13) }
14) if ($_POST['user_name']) {
15)   display_welcome();
16) }
17) else {
18)   display_empty_form();
19) }
```

Figure 11.5 Illustrating a function with arguments and return value

```
 0) function course_instructor ($course, $teaching_assignments) {
 1)   if (array_key_exists($course, $teaching_assignments)) {
 2)   $instructor = $teaching_assignments[$course];
 3)   RETURN "$instructor is teaching $course";
 4)   }
 5)   else {
 6)   RETURN "there is no $course course";
 7)   }
 8) }
 9) $teaching = array('Database' => 'Smith', 'OS' => 'Carrick',
                      'Graphics' => 'Kam');
10) $teaching['Graphics'] = 'Benson'; $teaching['Data Mining'] = 'Li';
11) $x = course_instructor('Database', $teaching);
12) print($x);
13) $x = course_instructor('Computer Architecture', $teaching);
14) print($x);
```

PHP Server Variables and Forms (1/2)

- Built-in entries
  - `$_SERVER` auto-global built-in array variable
  - Provides useful information about server where the PHP interpreter is running

PHP Server Variables and Forms (2/2)

- Examples:
  - **$_SERVER['SERVER_NAME']**
  - **$_SERVER['REMOTE_ADDRESS']**
  - **$_SERVER['REMOTE_HOST']**
  - **$_SERVER['PATH_INFO']**
  - **$_SERVER['QUERY_STRING']**
  - **$_SERVER['DOCUMENT_ROOT']**

- $_POST
  - Provides input values submitted by the user through HTML forms specified in <INPUT> tag

Overview of PHP Database Programming

- PEAR DB library
  - Part of PHP Extension and Application Repository (PEAR)
  - Provides functions for database access

Connecting to a Database

- Library module DB.php must be loaded
- DB library functions accessed using `DB::<function_name>`
- `DB::connect('string')`
  - Function for connecting to a database
  - Format for 'string' is: `<DBMS software>://<user account>:<password>@<database server>`

Figure 11.6 Connecting to a database, creating a table, and inserting a record

```
require 'DB.php';
$d = DB::connect('oci8://acct1:pass12@www.host.com/db1');
if (DB::isError($d)) { die("cannot connect - " . $d->getMessage());}
...
$q = $d->query("CREATE TABLE EMPLOYEE
  (Emp_id INT,
  Name VARCHAR(15),
  Job VARCHAR(10),
  Dno INT);" );
if (DB::isError($q)) { die("table creation not successful - " .
                        $q->getMessage()); }
  ...
$d->setErrorHandling(PEAR_ERROR_DIE);
...
$eid = $d->nextID('EMPLOYEE');
$q = $d->query("INSERT INTO EMPLOYEE VALUES
  ($eid, $_POST['emp_name'], $_POST['emp_job'], $_POST['emp_dno'])" );
...
$eid = $d->nextID('EMPLOYEE');
$q = $d->query('INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?)',
array($eid, $_POST['emp_name'], $_POST['emp_job'], $_POST['emp_dno']) );
```

Some Database Functions

- Query function
  - `$d->query` takes an SQL command as its string argument
  - Sends query to database server for execution
- `$d->setErrorHandling(PEAR_ERROR_DIE)`
  - Terminate program and print default error messages if any subsequent errors occur

Collecting Data from Forms and Inserting Records

- Collect information through HTML or other types of Web forms
- Create unique record identifier for each new record inserted into the database
- PHP has a function `$d->nextID` to create a sequence of unique values for a particular table
- **Placeholders**
  - Specified by `?` symbol

Retrieval Queries from Database Tables

- `$q`
  - Variable that holds query result
  - `$q->fetchRow()` retrieve next record in query result and control loop
- `$allresult = $d->getAll(query)`
  - Holds all the records in a query result in a single variable called `$allresult`

Figure 11.7 Illustrating database retrieval queries

```
 0) require 'DB.php';
 1) $d = DB::connect('oci8://acct1:pass12@www.host.com/dbname');
 2) if (DB::isError($d)) { die("cannot connect - " . $d->getMessage()); }
 3) $d->setErrorHandling(PEAR_ERROR_DIE);
    ...
 4) $q = $d->query('SELECT Name, Dno FROM EMPLOYEE');
 5) while ($r = $q->fetchRow()) {
 6)   print "employee $r[0] works for department $r[1] \n" ;
 7) }
    ...
 8) $q = $d->query('SELECT Name FROM EMPLOYEE WHERE Job = ? AND Dno = ?',
 9)   array($_POST['emp_job'], $_POST['emp_dno']) );
10) print "employees in dept $_POST['emp_dno'] whose job is
      $_POST['emp_job']: \n"
11) while ($r = $q->fetchRow()) {
12)   print "employee $r[0] \n" ;
13) }
    ...
14) $allresult = $d->getAll('SELECT Name, Job, Dno FROM EMPLOYEE');
15) foreach ($allresult as $r) {
16)   print "employee $r[0] has job $r[1] and works for department $r[2] \n" ;
17) }
    ...
```

Other techniques (1/2)

- PHP runs on server
  - Sends HTML to client
- Many other languages/technologies for Web Db programming
- Examples:
- Java servlets:
  - Java objects on server, interact with client
  - Store information about interaction session

Other techniques (2/2)

- Java Server Pages (JSP)
  - Creates dynamic Web pages through scripting at server to send to client (somewhat like PHP)
- JavaScript
  - Scripting language, can run at client or server
- Java Script Object Notation (JSON):
  - Text-based representation of objects
  - Similar function to XML
  - Used in many NOSQL systems

Summary

- A Simple PHP Example
- Overview of Basic Features of PHP
- Overview of PHP Database Programming

Agenda

1. Session Overview
2. Basic SQL
3. Advanced SQL
4. Introduction to SQL Programming
5. Web Database Programming Using PhP
6. Summary and Conclusion

Summary

- Basic SQL
- Advanced SQL
- Introduction to SQL Programming
- Web Database Programming Using PhP

Assignments & Readings

- Readings
  - Slides and Handouts posted on the course web site
  - Textbook: Chapters 6, 7, 10, and 11

- Assignment #6:
  - Textbook exercises: 6.13, 6.14, 6.15, 7.5, 7.8, 7.9
  - Programming exercises: 10-7, 10-8, 10-11, 10-12, 10-13, 11.11, and 11.12 (repeat only 10-7 in exercises 10-11 and 10-12)

- Database Project Part I – Data Modeling (continued)

Next Session: Functional Dependencies and Normalization

- Refining a relational implementation, including the normalization process and the algorithms to achieve normalization

Any Questions?