SQL

- Structured Query Language
- Declarative
  - Specify the properties that should hold in the result, not how to obtain the result
  - Complex queries have procedural elements
- International Standard
  - SQL1 (1986)
  - SQL2 (SQL-92)
  - SQL3 (pieces have started to appear; also known as SQL-99)
- Two components
  - DDL statements
  - DML statements

SQL DDL Statements

- Create schema
  ```
  CREATE SCHEMA Schema_Name AUTHORIZATION User_Name
  ```
- Create table
  - Specify a new relation scheme
  - General form
  ```
  CREATE TABLE <Table_Name>
  (Attribute_1 <Type>[DEFAULT <value>][<Null constraint>],
   Attribute_2 <Type>[DEFAULT <value>][<Null constraint>],
   ...
   Attribute_n <Type>[DEFAULT <value>][<Null constraint>],
  [<<Constraints>>])
  ```
Example

- Design
  - **Emp** (Eno, Ename, Title, City)
  - **Project** (Pno, Pname, Budget, City)
  - **Pay** (Title, Salary)
  - **Works** (Eno, Pno, Resp, Dur)

- Definition of **Project**
  ```sql
  CREATE TABLE Project
  (Pno CHAR(3),
   Pname VARCHAR(20),
   Budget DECIMAL(10,2) DEFAULT 0.00,
   City CHAR(9));
  ```

- Assume others are defined similarly.

Allowable Data Types

- Numeric
  - INT, SHORTINT
  - REAL (FLOAT), DOUBLE PRECISION
  - DECIMAL(i,j)

- Character-string
  - CHAR(n), VARCHAR(n)

- Bit-string
  - BIT(n), BIT VARYING(n)

- Date
  - YYYY-MM-DD

- Time
  - HH:MM:SS

- Timestamp
  - both DATE and TIME fields plus a minimum of six positions for fractions of seconds
User-Defined Types

■ Create a DOMAIN
  `CREATE DOMAIN <domain_name> AS <primitive_type>`

■ Example
  `CREATE DOMAIN Gender AS CHAR(1)`

■ Generally useful only for easy modification of type specification.

■ The value can be defaulted by the DEFAULT specification

Attribute Constraints

■ Not null
  • Specifies that an attribute cannot contain null values

■ Unique
  • Specifies that an attribute cannot contain duplicates

■ Primary key
  • Designates a set of columns as the table’s primary key
  • Implies UNIQUE and NOT NULL

■ Foreign key
  • Designates a set of columns as the foreign key in a referential constraint
Example

■ Given
  Emp (Eno, Ename, Title, City)
  Project(Pno, Pname, Budget, City)
  Pay(Title, Salary)
  Works(Eno, Pno, Resp, Dur)
■ Enhance the previous definition of Project:

```sql
CREATE TABLE Project
(Pno CHAR(3),
 Pname VARCHAR(20),
 Budget DECIMAL(10,2) DEFAULT 0.00
 City CHAR(9));
PRIMARY KEY (PNO));
```

Referential Constraints

■ REFERENTIALLY TRIGGERED ACTION
  ▪ Referential integrity: A key of one relation appears as an attribute (foreign key) of another relation.
  ▪ Example:
    Emp (Eno, Ename, Title, City)
    Pay(Title, Salary)
  ▪ Deletion or update of primary key tuple requires action on the foreign key tuple. Specify constraint on delete or update.
  ▪ How to manage?
    ➤ reject
    ➤ cascade: (on delete) automatically remove foreign keys if a referenced key is removed or (on update) change the foreign key value to the new value of the referenced key
    ➤ set null
    ➤ set default
Example

Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Works(Eno, Pno, Resp, Dur)

- Definition of deposit

```sql
CREATE TABLE Works
(Eno CHAR(3),
Pno CHAR(3),
Resp CHAR(15),
Dur INT,
PRIMARY KEY (Eno, Pno),
FOREIGN KEY (Eno) REFERENCES Emp(Eno)
ON DELETE SET NULL
ON UPDATE CASCADE,
FOREIGN KEY (Pno) REFERENCES Project(Pno));
```

Other SQL DDL Commands

- DROP SCHEMA
  - Delete an entire schema
  - CASCADE: delete all the tables in the schema
  - RESTRICT: delete only if empty

- DROP TABLE
  - RESTRICT: delete only if not referenced in a constraint

- ALTER TABLE
  - change definition
SQL Queries

■ Basic SQL structure

SELECT  \( A_1, A_2, \ldots, A_n \) 
FROM  \( R_1, R_2, \ldots, R_m \) 
WHERE  \( P \)

where

\( A_i \) are attributes from…

\( R_i \) which are relations

\( P \) is a predicate

* can replace \( A_i \)’s if all attributes are to be retrieved

What Kind of Predicates?

■ Simple predicates:

- Expression \( \theta \) Value where Expression can be an attribute or an arithmetic expression involving attributes
  \( \theta = \{<, >, =, <=, >=, <>\} \) and Value can be from one of the data types

- Example:
  - Name = ‘J. Doe’
  - \( (Age + 30) \geq 65 \)

■ Compound predicates

- Simple predicates combined with logical connectives
  
  \( \text{AND, OR, NOT} \)
Example Simple Queries

Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Pay(Title, Salary)
Works(Eno, Pno, Resp, Dur)

- List names of all employees.
  ```
  SELECT Ename
  FROM Emp
  ```

- List names of all projects together with their budgets.
  ```
  SELECT Pname, Budget
  FROM Project
  ```

- Find all cities where at least one project exists.
  ```
  SELECT DISTINCT City
  FROM Project
  ```

Queries With Predicates

Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Pay(Title, Salary)
Works(Eno, Pno, Resp, Dur)

- Find all professions that make more than $50,000.
  ```
  SELECT Title
  FROM Pay
  WHERE Salary > 50000
  ```

- Find all employees who work on a project as managers for longer than 17 months.
  ```
  SELECT Eno
  FROM Works
  WHERE Dur > 17 AND Resp='Manager'
  ```
Ordering the Results

Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Pay(Title, Salary)
Works(Eno, Pno, Resp, Dur)

- Find the names and budgets of all projects with budget greater than $250,000 and order the result in ascending order of budget values.

```
SELECT Pname, Budget
FROM Project
WHERE Budget > 250000
ORDER BY Budget
```

- Default is ascending order, but descending order can be specified by the DESC keyword.

Queries Over Multiple Relations

Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Pay(Title, Salary)
Works(Eno, Pno, Resp, Dur)

- List the name and titles of all employees who work on a project for more than 17 months.

```
SELECT Ename, Title
FROM Emp, Works
WHERE Dur > 17
AND Emp.Eno = Works.Eno
```

- Find the name and titles of all employees who work on a project located in Waterloo.

```
SELECT Ename, Title
FROM Emp E, Works W, Project P
WHERE P.City = 'Waterloo'
AND E.Eno = W.Eno
AND W.Pno = P.Pno
```
Queries Over Multiple Relations

**Emp** *(Eno, Ename, Title, City)*
**Project** *(Pno, Pname, Budget, City)*
**Pay** *(Title, Salary)*
**Works** *(Eno, Pno, Resp, Dur)*

- List the pairs of employees and projects that are co-located.
  
  ```sql
  SELECT Eno, Pno
  FROM Emp, Project
  WHERE Emp.City = Project.City
  ```

- List the pairs of employee names who are located in the same city.
  
  ```sql
  SELECT E1.Ename, E2.Ename
  FROM Emp E1, Emp E2
  WHERE E1.City = E2.City
  ```

Queries Involving Set Operators

- **UNION, EXCEPT (MINUS), INTERSECT**

  - Operands may be specified by
    - create temp relations
      ```sql
      ASSIGN TO temp-1
      SELECT A_1, \ldots, A_n
      FROM R_1, \ldots, R_m
      WHERE P;
      ```
    - use parentheses
      ```sql
      (SELECT A_1, \ldots, A_n
      FROM R_1, \ldots, R_m
      WHERE P)
      UNION
      (SELECT B_1, \ldots, B_p
      FROM S_1, \ldots, S_r
      WHERE Q);
      ```
Queries With Set Operators

Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Pay(Title, Salary)
Works(Eno, Pno, Resp, Dur)

- Find all cities where there is either an employee or a project.
  \[
  \text{(SELECT City FROM Emp) UNION (SELECT City FROM Project)}
  \]

- Find all cities in which an employee works but no projects are located.
  \[
  \text{(SELECT City FROM Emp) EXCEPT (SELECT City FROM Project)}
  \]

Queries With Set Operators

Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Pay(Title, Salary)
Works(Eno, Pno, Resp, Dur)

- Find all cities where there is both an employee and a project.
  \[
  \text{(SELECT City FROM Emp) INTERSECT (SELECT City FROM Project)}
  \]

- List names of all projects and employees in Waterloo.
  \[
  \text{(SELECT Ename FROM Emp WHERE City = 'Waterloo') UNION ALL (SELECT Pname FROM Project WHERE City = 'Waterloo')}\]

4-19

4-20
Queries With Nested Structures

- Queries within the WHERE clause of an outer query
  
  ```sql
  SELECT
  FROM
  WHERE OPERATOR (
  SELECT
  FROM
  WHERE
  )
  ```

- There can be multiple levels of nesting
- These can usually be written using other constructs (UNION, JOIN, etc).
- Three operators: IN, (NOT) EXISTS, [CONTAINS]

"IN" Construct

```sql
SELECT
FROM
WHERE A_1, ..., A_n IN
  (SELECT A_1, ..., A_n
   FROM R_1, ..., R_m
   WHERE P)
```

- Semantics: Tuples with attributes $A_1, ..., A_n$ are found in the relation that is returned as a result of the calculation of the inner query.
- Other comparison operators {$<$, $<=$, $>$, $>=$, $<>$} can be used with ALL or with ANY in place of IN.
“IN” Construct Example

Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Pay(Title, Salary)
Works(Eno, Pno, Resp, Dur)

Find names of all employees who work in cities where projects with budgets less than $50,000 are located.
SELECT Ename
FROM Emp
WHERE City IN (SELECT City
FROM Project
WHERE Budget < 50000)

Find names of all employees who work in a city where a project is located or are in a profession that pays more than $60,000.
SELECT Ename
FROM Emp
WHERE City IN (SELECT City
FROM Project )
OR Title IN (SELECT Title
FROM Pay
WHERE Salary > 60000)
“IN” Construct Example

Find the names of all the projects that have budgets greater than all projects in Calgary.

\[
\text{SELECT Pname FROM Project WHERE Budget > ALL (SELECT Budget FROM Project WHERE City = 'Calgary')}
\]

One can use ANY if what is desired is to find projects whose budget is greater than some project in Calgary.

“EXISTS” Construct

\[
\text{SELECT FROM WHERE (NOT) EXISTS (SELECT * FROM R_1, ..., R_m WHERE P)}
\]

Semantics: For each tuple of the outer query, execute the inner query; if there is at least one (no) tuple in the result of the inner query, then retrieve that tuple of the outer query.

This accounts for the “there exists” type of queries.
“EXISTS” Construct Example

**Emp** (Eno, Ename, Title, City)  
**Project** (Pno, Pname, Budget, City)  
**Pay** (Title, Salary)  
**Works** (Eno, Pno, Resp, Dur)

- Find the names of employees who work in a city in which some project is located.
  
  ```sql
  SELECT Ename  
  FROM Emp  
  WHERE EXISTS (SELECT *  
                FROM Project  
                WHERE Emp.City=Project.City)
  ```

- Find the names of employees who work in a city in which no project is located.
  
  ```sql
  SELECT Ename  
  FROM Emp  
  WHERE NOT EXISTS (SELECT *  
                    FROM Project  
                    WHERE Emp.City=Project.City)
  ```

- Find the names and titles of all the employees who do not work in a project.
  
  ```sql
  SELECT Ename, Title  
  FROM Emp  
  WHERE NOT EXISTS (SELECT *  
                    FROM Works  
                    WHERE Emp.Eno = Works.Eno)
  ```
“EXISTS” Construct Example

**Emp** (Eno, Ename, Title, City)
**Project** (Pno, Pname, Budget, City)
**Pay** (Title, Salary)
**Works** (Eno, Pno, Resp, Dur)

- Find all the employees who work on every project.
  - Find all employees such that there is no project on which they do not work
  
  ```sql
  SELECT *
  FROM Emp
  WHERE NOT EXISTS
    (SELECT Pno
     FROM Project
     WHERE NOT EXISTS
       (SELECT *
        FROM Works
        WHERE Project.Pno=Works.Pno
        AND Emp.Eno = Works.Eno))
  ```

Tuple Calculus and SQL

- Basic SQL Query:
  ```sql
  SELECT A_1, A_2, ..., A_n
  FROM R_1, R_2, ..., R_m
  WHERE P(R_1, R_2, ..., R_m)
  ```

- Tuple Calculus
  ```
  \{t[A_1],..., t[A_n] | \exists r_1 \in R_1, ..., \exists r_k \in R_k
  \land (\land_{i=1}^{n} t(A_i) = r_i(A_i)) \land P(r_1, ..., r_k)\}
  ```

- Note: Basic SQL query uses only \(\exists\); No explicit construct for \(\forall\)
Tuple Calculus & SQL

Example: “Find all the employees who work on every project.”

Tuple Calculus
\[
\{ e \mid e \in \text{Emp} \land \forall p \in \text{Project} (\exists w \in \text{Works} \\
(p[\text{Pno}] = w[\text{Pno}] \land w[\text{Pno}] = e[\text{Pno}])) \}
\]

Eliminate \( \forall \): \( \forall x F(x) \equiv \neg \exists x \neg F(x) \)
\[
\{ e \mid e \in \text{Emp} \land \neg \exists p \in \text{Project} \neg (\exists w \in \text{Works} \\
(p[\text{Pno}] = w[\text{Pno}] \land w[\text{Pno}] = e[\text{Pno}])) \}
\]

Tuple Calculus & SQL

Convert to SQL Query
- Basic Rule: One level of nesting for each “\( \neg \exists \)”
- The corresponding SQL query becomes:

```sql
SELECT *
FROM Emp
WHERE NOT EXISTS
(SELECT Pno
FROM Project
WHERE NOT EXISTS
(SELECT *
FROM Works
WHERE Project.Pno=Works.Pno
AND Emp.Eno = Works.Eno))
```
“CONTAINS” Construct

\[
\begin{align*}
\text{(SELECT} & \quad A_1, \ldots, A_n \\
\text{FROM} & \quad R_1, \ldots, R_m \\
\text{WHERE} & \quad P) \\
\text{CONTAINS} & \quad \text{(SELECT} \quad B_1, \ldots, B_p \\
\text{FROM} & \quad S_1, \ldots, S_r \\
\text{WHERE} & \quad Q) 
\end{align*}
\]

Semantics: Compare the result relations of the two queries and return TRUE if the second one is a subset of the first.

“CONTAINS” Construct Example

Emp (Eno, Ename, Title, City)  
Project(Pno, Pname, Budget, City)  
Pay(Title, Salary)  
Works(Eno, Pno, Resp, Dur)

Find all the employees who work on all the projects located in Edmonton.

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Emp} \\
\text{WHERE} & \quad (\text{SELECT} \quad \text{Pno} \\
& \quad \text{FROM} \quad \text{Works} \\
& \quad \text{WHERE} \quad \text{Emp.Eno} = \text{Works.Eno}) \\
\text{CONTAINS} & \quad (\text{SELECT} \quad \text{Pno} \\
& \quad \text{FROM} \quad \text{Project} \\
& \quad \text{WHERE} \quad \text{City} = \text{‘Edmonton’})
\end{align*}
\]

4-34
Aggregate Functions

- Specify a function that calculates a numeric value from a given relation.
  - The function is usually applied to an attribute.
  - COUNT, SUM, MAX, MIN, AVG

```
SELECT AggFunc(A_i), ..., AggFunc(A_j)
FROM R_1, ..., R_m
WHERE P
```

Aggregate Query Examples

- Find the total budgets of projects in Waterloo.
  ```
  SELECT SUM(Budget)
  FROM Project
  WHERE City = 'Waterloo'
  ```

- Find the number of cities where there is a project on which employee E4 works.
  ```
  SELECT COUNT(DISTINCT City)
  FROM Project, Works
  WHERE Project.Pno = Works.Pno
  AND Works.Eno = E4'
  ```
Aggregate Query Examples

**Emp** (Eno, Ename, Title, City)
**Project** (Pno, Pname, Budget, City)
**Pay** (Title, Salary)
**Works** (Eno, Pno, Resp, Dur)

- Find the names of projects which have budgets greater than the average budget of all the projects.

```sql
SELECT Pname
FROM Project
WHERE Budget >
(SELECT AVG(Budget)
FROM Project)
```

Grouping Queries

- Group the results according to a set of attributes

```sql
SELECT A_1, ..., A_n
FROM R_1, ..., R_m
WHERE p
GROUP BY A_j, ..., A_k
```

- Rules:
  - All of the attributes in the SELECT clause that are not involved in an aggregation operation have to be included in the GROUP BY clause.
  - GROUP BY can have more attributes ($k \geq n$)
Predicates on Groups

- Group the results according to a set of attributes if they satisfy a certain condition

\[
\begin{align*}
\text{SELECT} & \quad A_1, \ldots, A_n \\
\text{FROM} & \quad R_1, \ldots, R_m \\
\text{WHERE} & \quad P \\
\text{GROUP BY} & \quad A_j, \ldots, A_k \\
\text{HAVING} & \quad Q
\end{align*}
\]

- Rules:
  - $Q$ must have a single value per group.
  - An attribute in $Q$ has to either appear in an aggregation operator or be listed in the GROUP BY

Grouping Query Examples

**Emp** (Eno, Ename, Title, City)
**Project** (Pno, Pname, Budget, City)
**Pay** (Title, Salary)
**Works** (Eno, Pno, Resp, Dur)

- Find the cities in which more than 2 employees live.

\[
\begin{align*}
\text{SELECT} & \quad \text{City} \\
\text{FROM} & \quad \text{Emp} \\
\text{GROUP BY} & \quad \text{City} \\
\text{HAVING} & \quad \text{COUNT}(\ast) > 2
\end{align*}
\]

- Find the projects on which more than 2 employees share a responsibility.

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad \text{Pno} \\
\text{FROM} & \quad \text{Works} \\
\text{GROUP BY} & \quad \text{Pno, Resp} \\
\text{HAVING} & \quad \text{COUNT}(\ast) > 2
\end{align*}
\]
Grouping Query Examples

- For each city where there are more than three projects, find the names of projects.

```sql
SELECT City, Pname
FROM Project
WHERE City IN
(SELECT City
FROM Project
GROUP BY City
HAVING COUNT(*) > 3)
```

Grouping Query Examples

- Find the cities and the total budget where the average project budget is greater than $120,000.

```sql
SELECT City, SUM(Budget)
FROM Project
GROUP BY City
HAVING AVG(Budget) > 120000
```
Grouping Query Examples

- **Emp** *(Eno, Ename, Title, City)*
- **Project** *(Pno, Pname, Budget, City)*
- **Pay** *(Title, Salary)*
- **Works** *(Eno, Pno, Resp, Dur)*

For each project that employs more than 2 programmers, list the project number and the average duration of assignment of the programmers.

```sql
SELECT Pno, AVG(Dur)
FROM Works
WHERE Resp = 'Programmer'
GROUP BY Pno
HAVING COUNT(*) > 2
```

Update Commands

- Assume \( R(A_1, \ldots, A_n) \)
- **INSERT**
  - Form
    ```sql
    INSERT INTO R
    VALUES (value(A_1), \ldots, value(A_n))
    ```
  - Explicitly specify attribute names
    ```sql
    INSERT INTO R(A_{i_1}, \ldots, A_{k})
    VALUES (value(A_{i_1}), \ldots, value(A_{k}))
    ```
  - Insert a set of tuples as well
    ```sql
    INSERT INTO R (QUERY)
    ```
Update Command Examples

- **Emp** (Eno, Ename, Title, City)
- **Project** (Pno, Pname, Budget, City)
- **Pay** (Title, Salary)
- **Works** (Eno, Pno, Resp, Dur)

1. Insert a new employee record for John Smith who is assigned number E24, is a programmer and works in Waterloo.

   \[
   \text{INSERT INTO Emp(Eno, Ename, Title, City)} \\
   \text{VALUES ('E24','John Smith','Programmer','Waterloo')}
   \]

2. Insert into Pay tuples for titles that exist in Emp; set salary to $0.

   \[
   \text{INSERT INTO Pay} \\
   \text{(SELECT Title, 0} \\
   \text{FROM Emp)}
   \]

Update Commands

- **DELETE**

  \[
  \text{DELETE FROM } R \\
  \text{WHERE } P
  \]

- Delete all the employees who have worked on project P3 for less than 3 months.

  \[
  \text{DELETE FROM Emp} \\
  \text{WHERE Eno IN} \\
  \text{(SELECT Eno} \\
  \text{FROM Works} \\
  \text{WHERE Dur < 3)}
  \]
Update Commands

- Assume $R(A_1, \ldots, A_n)$
- UPDATE

```
UPDATE R
SET $A_i=value$, ..., $A_k=value$
WHERE $P$
```

Update Command Examples

**Emp** ($Eno$, $Ename$, $Title$, $City$)
**Project** ($Pno$, $Pname$, $Budget$, $City$)
**Pay** ($Title$, $Salary$)
**Works** ($Eno$, $Pno$, $Resp$, $Dur$)

- Increase by 5% the budgets of projects that are located in Edmonton and employ 3 or more people.

```
UPDATE Project
SET $Budget = Budget*1.05$
WHERE $City = ‘Edmonton’$
AND $Pno$ IN
(SELECT $Pno$
FROM Works
GROUP BY $Pno$
HAVING COUNT(*) $>= 3$)
```
View Definition

- General form

```
CREATE VIEW V[ (A₁, ..., Aₖ) ]
AS SELECT A₁, ..., Aₙ
FROM R₁, ..., Rₘ
WHERE P
```

- In queries, treat view V as any base relation

```
SELECT A_j, ..., A_l
FROM V
WHERE Q
```

- Updates of views may be restricted

View Definition Example

```
Emp (Eno, Ename, Title, City)
Project(Pno, Pname, Budget, City)
Pay(Title, Salary)
Works(Eno, Pno, Resp, Dur)

Create a view Waterloo-Projects of projects that are located in Waterloo.
CREATE VIEW waterloo-projects
AS SELECT *
FROM Project
WHERE City = 'Waterloo'

Create a view Rich-Employees consisting of employees who make more than $75,000.
CREATE VIEW rich-employees
AS SELECT Eno, Ename, Title, City
FROM Emp, Pay
WHERE Salary > 75000
AND Emp.Title = Pay.Title
```
View Definition Example

- **Emp** (Eno, Ename, Title, City)
- **Project** (Pno, Pname, Budget, City)
- **Pay** (Title, Salary)
- **Works** (Eno, Pno, Resp, Dur)

Create a view project-employ that gives, for each project at each city, the number of employees who work on the project and their total employment duration.

```
CREATE VIEW project-employ (Name, City, Number, Total)
AS SELECT Pname, City, COUNT(Eno), SUM(Dur)
FROM Project P, Works W
WHERE P.Pno = W.Pno
GROUP BY City, Pname
```

View Update

- A view with a single defining table is updatable if the view attributes contain the primary key or some other (non-null) candidate key.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using aggregate functions are not updatable.
View Management

- Two strategies for view management
  - Query modification
    - A view is a virtual relation
    - Multiple queries on complex views
    - Query processor modifies a query on a view with the view definition
  - View materialization
    - View is a materialized table in the database
    - Incrementally update the view (update propagation)

Handling Nulls

- Attribute values can be NULL
- NULL is not a constant, nor can it be used as an operand
  - NAME = NULL Wrong!
  - NULL + 30 Wrong!
- Operating on variables with NULL values
  - If x is NULL
    - x <arithmetic op.> constant is NULL
    - x <arithmetic op.> y is NULL
  - Comparing a NULL value with any other value returns UNKNOWN (three-valued logic).
  - In SQL, UNKNOWN is sometimes treated as false (SELECT) and sometimes as true (constraints).
Outer Join

- Ensures that tuples from one or both relations that do not satisfy the join condition still appear in the final result with other relation’s attribute values set to NULL
- Extend SQL2 join expression
  - R \{NATURAL\} JOIN S [ON <condition>]
  - to provide
    - R \{NATURAL\} FULL OUTER JOIN S [ON <condition>]
    - R \{NATURAL\} LEFT OUTER JOIN S [ON <condition>]
    - R \{NATURAL\} RIGHT OUTER JOIN S [ON <condition>]

Example: Find the employees and the projects that they work on.
- Emp NATURAL LEFT OUTER JOIN Works
- Produces employees even if they do not work on any project at the moment.

Embedded SQL

- Ability to access a database from within an application program.
- Why?
  - SQL is not sufficient to write general applications
  - SQL has bindings for various programming languages.
    - C, C++, Java
    - Bindings describe how applications written in these host languages can interact with a DBMS.
Embedded SQL Application Development

Source Files with Embedded SQL

Preprocessor

Modified Source Files

Host Language Compiler

Object Code Files

Host Language Linker

Executable

Libraries

Other Object Files & Libraries

Embedded SQL Issues

- **Interface**
  - The same as ad hoc SQL but with `EXEC SQL` command

- **Using query results within the application program**
  - Define shared variables using the format allowed in host language
  - Shared variables can be used in `INSERT`, `DELETE`, `UPDATE` as well as regular queries and schema modification statements
  - Retrieval queries require care
    - If the result relation has a single tuple: Each value in the `SELECT` statement requires a shared variable
    - If the result relation has a set of tuples: Cursor has to be defined
Example for Update

Consider an example to transfer some amount from one project’s budget to another project’s budget.

```c
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main() {
    EXEC SQL WHENEVER SQLERROR GOTO error;
    EXEC SQL CONNECT TO Company;
    EXEC SQL BEGIN DECLARE SECTION;
        int pno1[3], pno2[3]; /* two project numbers */
        int amount; /* amount to be transferred */
    EXEC SQL END DECLARE SECTION;
    /* Code (omitted) to read the project numbers and amount */
    EXEC SQL UPDATE Project
    SET Budget = Budget + :amount
    WHERE Pno = :pno2;
    EXEC SQL UPDATE Project
    SET Budget = Budget - :amount
    WHERE Pno = :pno1;
    EXEC SQL COMMIT RELEASE;
    return(0);
    error:
    printf("update failed, sqlcode = %ld\n", SQLCODE);
    EXEC SQL ROLLBACK RELEASE;
    return(-1);
}
```

Termination of SQL Queries

- **COMMIT**
  - If you want to make your update results permanent.
  - Embedded: `EXEC SQL COMMIT;`

- **ROLLBACK**
  - If you want to discard your results.
  - Embedded: `EXEC SQL ROLLBACK;`

We will study the full semantics of these commands later when we look at transactions.
Retrieval Queries

- If the result is only one tuple, follow the same approach.
- Example: For a given title, retrieve the salary.

```sql
EXEC SQL BEGIN DECLARE SECTION;
  char title[15]; /* title to be input by user */
  real sal; /* salary */
EXEC SQL END DECLARE SECTION;

/* Code (omitted) to get the title from user */
EXEC SQL SELECT Salary
  INTO :sal
  FROM Pay
  WHERE Title = :title;
/* Code (omitted) to print the result */
EXEC SQL END DECLARE SECTION;
```

More General Retrieval Queries

```sql
EXEC SQL BEGIN DECLARE SECTION;
  <shared variable declarations>
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE <cursor-name> [options] CURSOR FOR
  <query> [options]
EXEC SQL OPEN <cursor-name>

while(condition) {
  EXEC SQL FETCH FROM <cursor-name> INTO <shared-variable(s)>
  if(tuple exists) process it
  else break
}
EXEC SQL CLOSE <cursor-name>
```
Cursors

- Use when the embedded SQL query is expected to return a relation with more than one tuple
- Think about it almost as a pointer to successive tuples in the result relation. At a given moment, it can be
  - Before the first tuple
  - On a tuple
  - After the last tuple

<table>
<thead>
<tr>
<th></th>
<th>Before 1st tuple</th>
<th>-n</th>
<th>-n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>(n-1)</td>
<td>(n+1)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>-n</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>n+1</td>
<td>After last tuple</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Use of Cursors

1. Declare the cursor
   - Associates a cursor identifier with a query
2. Open the cursor
   - Causes (conceptually) the query to be evaluated, generating a result
3. Fetch one or more tuples using the cursor
   - Each FETCH returns values from the tuple that the cursor is currently “pointing to”
4. Close the cursor
Cursor Declaration & Access

- **Declaration**

  ```sql
  EXEC SQL DECLARE <cursor-name>
  [INSENSITIVE] [SCROLL] CURSOR FOR <query>
  ORDER BY <attribute(s)>
  [FOR READ ONLY];
  ``

  **INSENSITIVE:** The cursor is insensitive to changes in the relations referred to in the query while the cursor is open.

  **SCROLL:** Allows the subsequent FETCH command to retrieve tuples other than the default which is moving forward.

  **ORDER BY:** Orders the results of the query according to some attribute(s).

  **FOR READ ONLY:** Ensures that accesses to the underlying relation(s) via this cursor will not change the relation.

- **Access**

  ```sql
  EXEC SQL FETCH [NEXT | PRIOR | FIRST | LAST | RELATIVE [+] [-] n] | ABSOLUTE [+] [-] n <cursor-name>
  [INTO <var1>, ..., <varn>]
  ```

Cursor Example

- For each project that employs more than 2 programmers, list the project number and the average duration of assignment of programmers.

  ```sql
  EXEC SQL BEGIN DECLARE SECTION;
  char pno[3]; /* project number */
  real avg-dur; /* average duration */
  EXEC SQL END DECLARE SECTION;
  ...
  EXEC SQL DECLARE duration CURSOR FOR
  SELECT Pno, AVG(Dur)
  FROM Works
  WHERE Resp = 'Programmer'
  GROUP BY Pno
  HAVING COUNT(*) > 2;
  ...
  EXEC SQL OPEN duration;
  ...
  while(1) {
    EXEC SQL FETCH FROM duration INTO :pno, :avg-dur
    if(strcmp(SQLSTATE, "02000") then break
    else print the info
  }
  EXEC SQL CLOSE duration
  ```
Dynamic (Embedded) SQL

- If the exact SQL statement to be executed is not known at the time the application is written, static embedding won’t work.
  - E.g., forms-based applications
- In this case, the query can either be input dynamically or it can be parameterized
  - Immediate execution
  - Preparation for later (and multiple) executions
  - Parameterization

Immediate Execution

- General form
  ```sql
  EXEC SQL EXECUTE IMMEDIATE :string
  ```
- Example
  ```sql
  EXEC SQL BEGIN DECLARE SECTION;
  char tup[] = "INSERT INTO EMP VALUES('E13', 'John Doe',...);"
  EXEC SQL END DECLARE SECTION;
  EXEC SQL EXECUTE IMMEDIATE :tup;
  ```
- Rules:
  - :string may not return an answer
  - :string may not contain parameters
  - Every time :string is executed, it is compiled ⇒ high overhead
Prepared Execution

- General form
  ```sql
  EXEC SQL PREPARE stmt FROM :string
  ```
- Rules
  - `:string` may return results (it may be a query)
  - `:string` may contain parameters
  - `stmt` is not a host variable, but an identifier of the statement used by the preprocessor
- Example
  ```sql
  EXEC SQL BEGIN DECLARE SECTION;
  char tup[] = "INSERT INTO EMP VALUES('E13', 'John Doe', ...)";
  EXEC SQL END DECLARE SECTION;
  EXEC SQL PREPARE S1 FROM :tup;
  EXEC SQL EXECUTE S1;
  ...
  EXEC SQL EXECUTE S1;
  ```

Parametric Statements

- It is possible to parameterize the strings in dynamic SQL statements
- Use placeholders (or parameter markers) - ? - where literals can appear (not in place of relation names, column names, etc.
  - ```sql
    INSERT INTO Emp VALUES (?, ?, ?, ?)
    ```
- Indicate that host variable values will replace the placeholders
  ```sql
  EXEC SQL EXECUTE S1 USING :eno, :ename, :title, :city
  ```
- USING cannot be used with EXECUTE IMMEDIATE; has to be used with previously prepared statements
Parametric Example - Update

EXEC SQL BEGIN DECLARE SECTION;
char tup[] = "INSERT INTO Emp VALUES (?,?,?,?);
char eno[3], ename[15], title[10], city[12];
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE S1 FROM :tup;
/*code (omitted) to read values into :eno, etc */
EXEC SQL EXECUTE S1 USING :eno, :ename, :title,
    :city;
...

Parametric Example - Retrieval
(Single Tuple)

EXEC SQL BEGIN DECLARE SECTION;
char tup[] = "SELECT Salary FROM Pay WHERE Title = '?");
char title[15]; /* title to be input by user */
real sal; /* salary */
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE S1 FROM :tup;
while (...) {
    /*code (omitted) to read title from user
        into :title */
    EXEC SQL EXECUTE S1 INTO :sal USING :title;
...
}
...
Note: DB2 does not allow this; use dynamic cursor (next slide)
Dynamic Cursors

- If the result is a relation with many tuples, then use dynamic cursors.
- Define dynamic cursors similar to their static counterparts, but use
  - USING to parameterize the input to the cursor query
  - INTO to hold the output parameters

```sql
EXEC SQL DECLARE cname CURSOR FOR stmt;
EXEC SQL OPEN cname USING :var1 [,..., :varn];
EXEC SQL FETCH cname INTO :out1 [,..., :outk];
EXEC SQL CLOSE cname;
```

Dynamic Cursor Example

```sql
EXEC SQL BEGIN DECLARE SECTION;
    char resp[10]; /* responsibility to be input */
    char pno[3]; /* project number */
    real avg-dur; /* average duration */
    char s[] = "SELECT Pno, AVG(Dur) FROM Works WHERE
              Resp = '?\' GROUP BY Pno, Eno HAVING COUNT(*) > 2";
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE S1 FROM :s
EXEC SQL DECLARE duration CURSOR FOR S1;
read into :resp
EXEC SQL OPEN duration USING :resp;
while(1) {
    EXEC SQL FETCH FROM duration INTO :pno, :avg-dur
    if(strcmp(SQLSTATE, "02000") then break
    else print the info
}
EXEC SQL CLOSE duration
```
DBMS-Independent Application Development

- **Call-Level Interface (CLI)**
  - Vendor-neutral ISO standard programming interface for relational database systems.
  - Based on ODBC

- **Open Database Connectivity (ODBC)**
  - Microsoft developed programming interface for relational DBMSs with SQL interface.

- **Java Database Connectivity (JDBC)**
  - Collection of Java classes that provide an ODBC/CLI-like programming environment

- These provide object code-level portability from one DBMS to another, while embedded SQL provides only source code-level portability

---

**ODBC Architecture**

![ODBC Architecture Diagram]

ODBC

- ODBC Driver
- ODBC Driver
- ODBC Driver

ODBC Manager

- DBMS#1
- DBMS#2
- Non-DBMS Data Source
Attribute & Domain Constraints

- **CHECK**
  - specifies the condition that each row has to satisfy
- Enhance the previous definition of Project:
  ```
  CREATE TABLE Project
  (Pno CHAR(3),
   Pname VARCHAR(20),
   Budget DECIMAL(10,2) DEFAULT 0.00
     CHECK (BUDGET >= 0),
   City CHAR(9));
  PRIMARY KEY (Pno));
  ```
- Can be specified on domains as domain constraints
  ```
  CREATE DOMAIN Gender AS CHAR(1)
  CHECK (VALUE IN ('F', 'M'));
  ```

Tuple Constraints

- Use CHECK command and specify condition
- The condition is checked every time a tuple is inserted or updated and the action rejected if the condition is false
- Example
  ```
  CREATE TABLE Works
  (Eno CHAR(3),
   Pno CHAR(3),
   Resp CHAR(15),
   Dur INT,
  PRIMARY KEY (Eno,Pno),
  FOREIGN KEY (Eno) REFERENCES Emp(Eno)
    ON DELETE SET NULL
    ON UPDATE CASCADE,
  FOREIGN KEY (Pno) REFERENCES Project(Pno),
  CHECK (NOT (PNO<'P5') OR Dur>18));
  ```
More Complex Tuple Constraints

- Example: No project can employ more than two employees who are assigned to work more than 48 months.

```
CREATE TABLE Works
(Eno CHAR(3),
Pno CHAR(3),
Resp CHAR(15),
Dur INT,
PRIMARY KEY (Eno, Pno),
FOREIGN KEY (Eno) REFERENCES Emp(Eno)
    ON DELETE SET NULL
    ON UPDATE CASCADE,
FOREIGN KEY (Pno) REFERENCES Project(Pno),
CHECK(3 > ALL
(_SELECT COUNT(Eno)
FROM Works
WHERE Dur > 48
GROUP BY Pno));
```

Assertions

- Global constraints that apply to multiple relations.
- General form
  - `CREATE ASSERTION name CHECK (condition)`
- Example: The total salaries of employees who work on project P5 cannot exceed $500,000.

```
CREATE ASSERTION salary-control CHECK
((SELECT SUM(Salary)
FROM Emp E, Pay P, Works W
WHERE W.Pno = 'P5'
AND W.Eno = E.Eno
AND E.Title = P.Title) <= 500000);
```
Triggers

A trigger is a procedure that is automatically invoked by the DBMS in response to changes to the database.

Three parts:
- **Event**
  - Change to the database that activates the trigger
- **Condition**
  - A test or a query that needs to be checked when the trigger is activated
  - For queries, condition is *true* if the query returns any result
- **Action**
  - The procedure that is executed when the trigger is activated and the condition is *true*

Trigger Issues

With respect to invoking statement
- Execution before or after
- Execution in place of
- At the end of the transaction within which the trigger is activated (deferred)
- Asynchronously within the context of a separate transaction

How many times executed?
- Statement-level: once per invoking statement
- Row-level: once per modified tuple