Fpt University

Chapter 6

THE DATABASE LANGUAGE SQL

Objectives

- Student can write a SQL script.
- Student can compose SQL queries using set (and bag) operators, correlated subqueries, aggregation queries.
- Student can manipulate proficiently on complex queries

Contents

- Integrity constraints
- Structure Query Language
  - DDL
  - DML
  - DCL (self studying)
  - Sub query

REVIEW

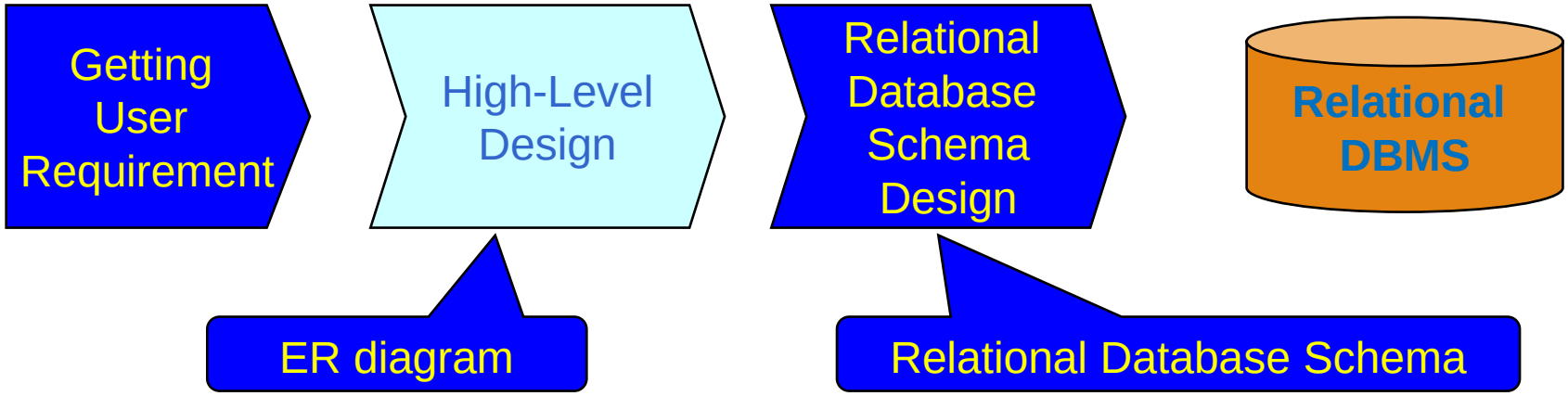


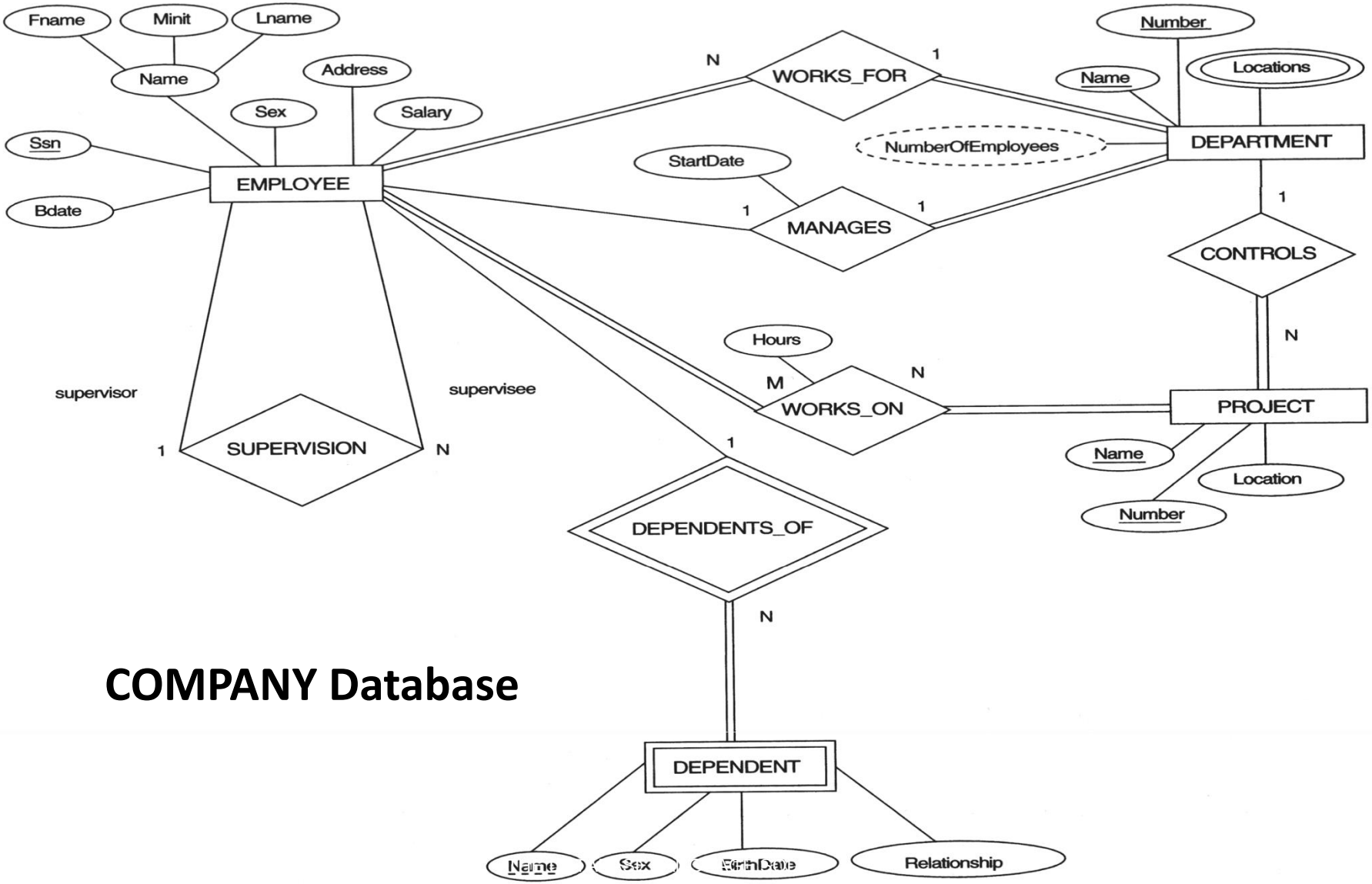
Figure 4.1: The database modeling and implementation process

Studied:

- ER diagram
- Relational model
- Convert ERD → Relational model

Now: we learn how to set up a relational database on DBMS

REVIEW – Entity Relationship Diagram



COMPANY Database

Integrity constraints

- Purpose: prevent semantic inconsistencies in data
- Kinds of integrity constraints:
  1. Key Constraints (1 table): Primary key, Candidate key (Unique)
  2. Attribute Constraints (1 table): NULL/NOT NULL; CHECK
  3. Referential Integrity Constraints (2 tables): FOREIGN KEY
  4. Global Constraints (n tables): CHECK or CREATE ASSERTION (self studying)

We will implement these constraints by SQL

Comparison of Strings

Two strings are equal (=) if they are the same sequence of characters

Other comparisons: <, >, ≤, ≤, ≠

Suppose $a=a_1a_2\ldots a_n$ and $b=b_1b_2\ldots b_m$ are two strings, the first is less than the second if $\exists$ $k \leq \min(n,m)$:

- $\forall i,\ 1 \leq i \leq k$: $a_i = b_i$, and
- $a_{k+1} < b_{k+1}$

Example

- ***fo****d****der*** < ***fo****o*
- ***bar*** < ***bar****g****ain***

Pattern Matching in SQL

Like or Not Like

```
SELECT
FROM
WHERE s LIKE p;
```

```
SELECT
FROM
WHERE s NOT LIKE p;
```

Two special characters

- **%** means any sequence of 0 or more characters
- _ means any one character

Pattern Matching in SQL

Example 5.1:

- Find all employees named as ‘Võ Việt Anh’

Example 5.2

- Find all employees whose name is ended at ‘Anh’

```
/*5.1*/
SELECT * FROM tblEmployee WHERE empName = N'Võ Việt Anh';
GO

/*5/2*/
SELECT * FROM tblEmployee WHERE empName LIKE N'%Anh';
GO
```

Pattern Matching in SQL

USING ESCAPE keyword

- SQL allows us to specify any one character we like as the escape character for a single pattern
- Example
  - WHERE s LIKE '%20!%%' ESCAPE !
  - Or WHERE s LIKE '%20@%%' ESCAPE @

➔ Matching any s string contains the 20% string

  - WHERE s LIKE 'x%%x%' ESCAPE x

➔ Matching any s string that begins and ends with the character %

Dates and Times

Dates and times are special data types in SQL

A *date* constant's presentation

- DATE '1948-05-14'

A *time* constant's presentation

- TIME '15:00:02.5'

A combination of dates and times

- TIMESTAMP '1948-05-14 12:00:00'

Operations on date and time

- Arithmetic operations
- Comparison operations

Null Values

Null value: special value in SQL

Some interpretations

- *Value unknown*: there is, but I don't know what it is
- *Value inapplicable*: there is no value that makes sense here
- *Value withheld*: we are not entitled to know the value that belongs here

Null is not a constant

Two rules for operating upon a NULL value in WHERE clause

- Arithmetic operators on NULL values will return a NULL value
- Comparisons with NULL values will return UNKNOWN

The Truth-Value UNKNOWN

Truth table for True, False, and Unknown

We can think of TRUE=1; FALSE=0; UNKNOWN=1/2, so

- x AND y = MIN(x,y); x OR y = MAX(x, y); NOT x = 1-x

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

The Truth-Value Unknown

SQL conditions in Where clause produce three truth values: True, False, and Unknown

Those tuples which condition has the value True become part of the answer

Those tuples which condition has the value False or Unknown are excluded from the answer

SQL Overview

- SQL (sequel) is a database language designed for managing data in relational database management systems, and originally based upon relational algebra.
- There are many different dialects of SQL
  - Ansi SQL (or SQL-86), SQL-92, SQL-99
  - SQL:2003, SQL:2006, SQL:2008, SQL:2009
- **Transact-SQL** (**T-SQL**) is Microsoft's and Sybase's proprietary extension to SQL.
- **PL/SQL** (**Procedural Language/Structured Query Language**) is Oracle Corporation's procedural extension for SQL and the Oracle relational database.
- Today, SQL is accepted as the standard RDBMS language

Data Definition Language - CREATE

- Database schema

Simple syntax: **CREATE DATABASE** dbname

Full syntax: **https://docs.microsoft.com/en-us/sql/database**

- Relation schema ~ table

CREATE TABLE tableName
(
fieldname1 datatype [*integrity_constraint*s],
fieldname2 datatype [*integrity_constraint*s],
....
)

Full syntax: **https://docs.microsoft.com/en-us/sql/table**

Data Definition Language - Demo

```
CREATE DATABASE EmpManagement;

USE EmpManagement;

CREATE TABLE tblEmployee
    (IdEmp INT identity(1,1) PRIMARY KEY,
     EmpName nvarchar(50) NOT NULL,
     DOB date CHECK(year(getdate())-year(DOB)>=18),
     PhoneNo char(12) Unique,
     Addr nvarchar(50) DEFAULT N'Hồ Chí Minh'
    );
```

Data Definition Language – ALTER, DROP

- Used to modify the structure of table, database
  - Add more columns

ALTER TABLE tableName

ADD columnName datatype [constraint]

  - Remove columns

ALTER TABLE tableName

DROP columnName datatype [constraint]

  - Modify data type

ALTER TABLE tableName

ALTER columnName datatype [constraint]

Data Definition Language– ALTER, DROP

- Add/remove constraints

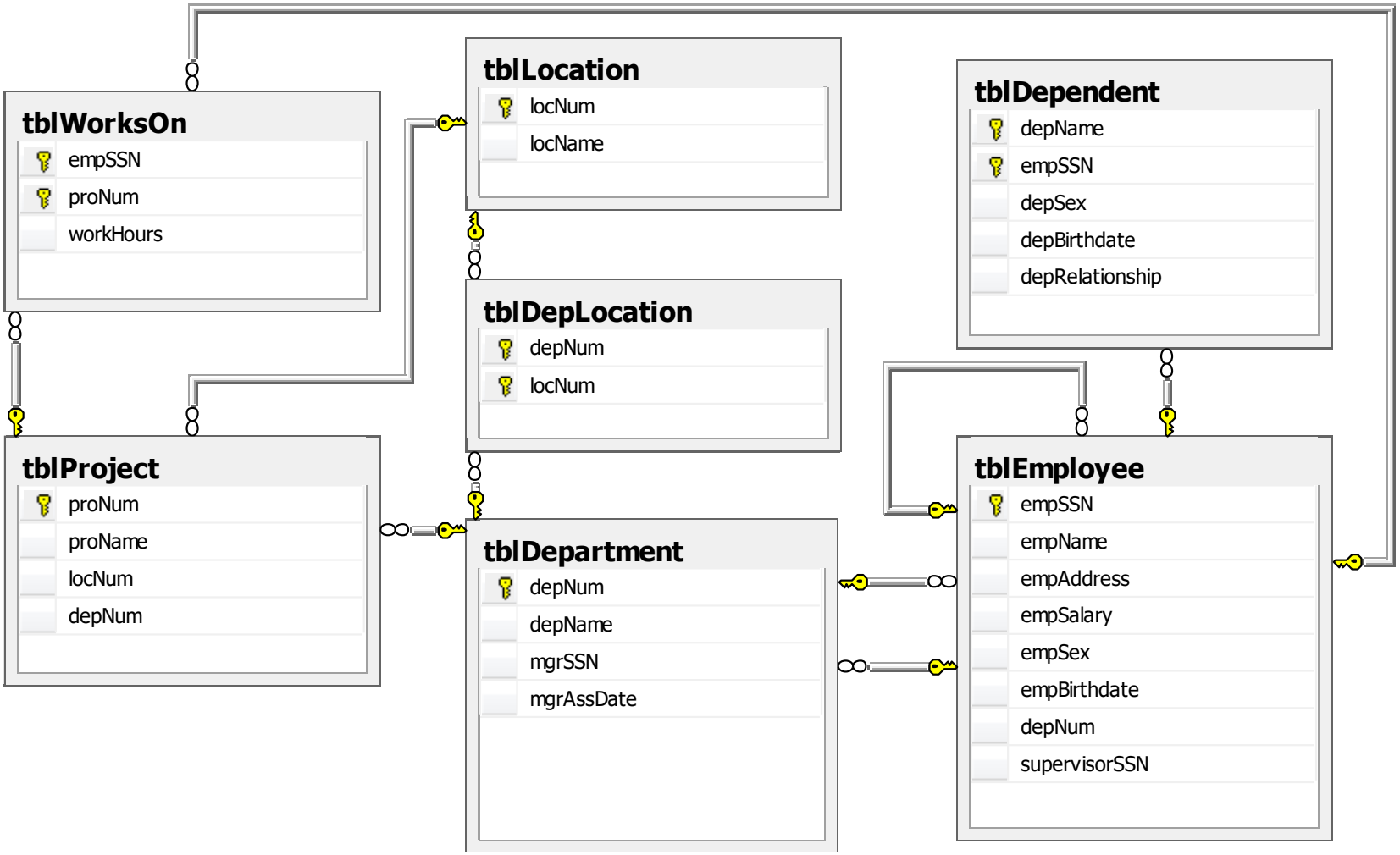
```
ALTER TABLE tablename
ADD CONSTRAINT constraintName PRIMARY KEY
(<attribute list>);

ALTER TABLE tablename
ADD CONSTRAINT constraintName FOREIGN KEY (<attribute list>)
REFERENCES parentTableName (<attribute list>);

ALTER TABLE tablename
ADD CONSTRAINT constraintName CHECK (expressionChecking)

ALTER TABLE tablename
DROP CONSTRAINT constraintName
```

Fpt University

Data Definition Language– ALTER, DROP

- **DROP TABLE** tableName
- **DROP DATABASE** dbName

Physical Diagram - FUHCompany



Data Manipulation Language (DML)

Key words: **INSERT, UPDATE, DELETE, SELECT**

INSERT INTO tableName
VALUES (<value 1>, ... <value n>)

INSERT INTO tableName(<listOfFields>)
VALUES (<value 1>, ... <value m>)

INSERT INTO tableName
SELECT listOfFields **FROM** another_tableName

```
/*22*/
INSERT INTO tblDepartment(depNum,depName)
VALUES(6, N'Phòng Kế Toán');

INSERT INTO tblDepartment
VALUES(7, N'Phòng Nhân Sự', NULL, NULL);
```

Data Manipulation Language (DML)

UPDATE tableName

SET columnName = newValue

[WHERE condition**]**

Note: newValue could be a value/ an expression/ a SQL statement

- Example: Update new salary and depNum for the employee named ‘Mai Duy An’

```
/*24*/
UPDATE   tblEmployee
SET      empSalary=empSalary+5000, depNum=2
WHERE    empName=N'Mai Duy An'
```

Data Manipulation Language (DML)

DELETE FROM tableName
[**WHERE** condition]

TRUNCATE TABLE tableName

- *What is difference between DELETE and TRUNCATE?*
- *What should we do before implement* **DELETE** *or* **TRUNCATE***? (referential integrity constraint)*
- *Example:*
  - remove a department named ‘Phòng Kế Toán’
  - remove a department which depNum is 7

```
/*22*/
DELETE
FROM    tblDepartment
WHERE   depName=N'Phòng Kế Toán'

DELETE
FROM    tblDepartment
WHERE   depNum=7
```

Data Manipulation Language (DML)

SQL Queries and Relational Algebra

SELECT *L*
FROM *R*
WHERE *C*

$\pi_L(\sigma_C(R))$

T-SQL : Basic Syntax for a simple SELECT queries

```
SELECT [ ALL | DISTINCT ]
        [ TOP n [ PERCENT ] ]
      * | {column_name | expression [alias],…}
[FROM table]
[WHERE conditions]
```

SELECT identifies *what* columns

- ALL: Specifies that duplicate rows can appear in the result set. ALL is the default
- DISTINCT: Specifies that only unique rows can appear in the result set. Null values are considered equal for the purposes of the DISTINCT keyword
- TOP *n* [PERCENT]:Specifies that only the first *n* rows are to be output from the query result set. *n* is an integer between 0 and 4294967295. If PERCENT is also specified, only the first *n* percent of the rows are output from the result set. When specified with PERCENT, *n* must be an integer between 0 and 100

FROM identifies *which* table

The WHERE clause follows the FROM clause. *Condition:* is composed of column names, expressions, constants, and a comparison operator

Common Query in SQL

Example 1: Listing all employees whose salary exceed at 50000

```
/*1*/
SELECT *
FROM tblEmployee
WHERE empSalary > 50000
```

- Example 2: Listing name and salary of all employees whose income exceed 50000

```
/*2*/
SELECT empName,empSalary
FROM tblEmployee
WHERE empSalary > 50000
```

Projection in SQL

Using alias name in select clause

Example 3:

- Listing full name and salary of all employees whose income exceed 50000

```
/*3*/
SELECT empName AS 'Họ và tên',empSalary AS 'Luong'
FROM tblEmployee
WHERE empSalary > 50000
```

Selection in SQL

Example 4

- List all under 40 year-old female or under 50 year-old male employees

```
/*4*/
SELECT empName AS 'Họ và tên',empSex AS 'Giới tính',
YEAR(GETDATE())-YEAR(empBirthdate) AS 'Tuổi'
FROM tblEmployee
WHERE (empSEX='F' AND YEAR(GETDATE())-YEAR(empBirthdate)<40)
OR (empSEX='M' AND YEAR(GETDATE())-YEAR(empBirthdate)<50)
```

Ordering the Output

Presenting the tuples produced by a query in sorted order

The order may be based on the value of any attribute

Syntax

SELECT *<list of attributes>*
FROM *<list of tables>*
WHERE *<conditions>*
ORDER BY *<list of attributes>*

Order by clause follows Where and any other clauses. The ordering is performed on the result of the From, Where, and other clauses, just before Select clause

Using keyword ASC for ascending order and DESC for descending order

Ordering the Output

Example 6:

- Listing all employee by department number ascreasingly, then by salary descreasingly

```
/*6*/
SELECT *
FROM tblEmployee
ORDER BY depNum ASC, empSalary DESC
GO
```

6.2 QUERIES INVOLVING MORE THAN ONE RELATION

Queries Involving More Than One Relation

SQL allows we combine two or more relations through joins, products, unions, intersections, and differences

Products and Joins in SQL

When data from more than one table in the database is required, a *join* condition is used.

Simple way to couple relations: list each relation in the From clause

Other clauses in query can refer to the attributes of any of the relations in the From clause

Products and Joins in SQL

Example 7:

- List all employees who work on ‘Phòng Phần mềm trong nước’ department

```
/*7*/
SELECT *
FROM tblEmployee E, tblDepartment D
WHERE e.depNum=d.depNum AND d.depName LIKE N'Phòng phần mềm trong nước';
GO
```

What we do if …

… a query involves several relations, and there are two or more attributes with the same name?

Tuple Variables

We may list a relation R as many times as we need

We use tuple variables to refer to each occurrence of R

Disambiguating Attributes

Example 8:

- Find all cities in which our company is

```
/*8*/
SELECT distinct l.locname
FROM tblLocation l, tblDepLocation d
WHERE l.locNum=d.locNum
GO
```

What we do if …

… a query involves two or more tuples from the same relation?

Example 9:

- Find all those project numbers which have more than two members

```
/*9*/
SELECT distinct w1.proNum as 'Project Number'
FROM tblWorksOn w1, tblWorksOn w2
WHERE w1.proNum=w2.proNum AND w1.empSSN <> w2.empSSN
GO
```

Union, Intersection, Difference of Queries

We combine relations using the set operations of relational algebra: union, intersection, and difference

SQL provides corresponding operators with UNION, INTERSECT, and EXCEPT for $\cup$, $\cap$, and -, respectively

Union, Intersection, Difference of Queries

Example 10.1

- Find all those employees whose name is begun by ‘H’ or salary exceed 80000

```
/*10.1*/
SELECT * FROM tblEmployee WHERE empName LIKE 'H%'
UNION
SELECT * FROM tblEmployee WHERE empSalary > 80000
GO
```

Union, Intersection, Difference of Queries

Example 10.2

- Find all those *normal* employees, that is who do not supervise any other employees

```
/*10.2*/
SELECT empSSN FROM tblEmployee
EXCEPT
SELECT supervisorSSN FROM tblEmployee
GO
```

Union, Intersection, Difference of Queries

Example 10.3

- Find all employees who work on projectB and projectC

```
/*10.3*/
SELECT empSSN
FROM tblWorksOn w, tblProject p
WHERE w.proNum=p.proNum AND p.proName='ProjectB'
INTERSECT
SELECT empSSN
FROM tblWorksOn w, tblProject p
WHERE w.proNum=p.proNum AND p.proName='ProjectC'
GO
```

6.3 SUB QUERIES

Sub-queries

One query can be used to help in the evaluation of another

A query that is part of another is called a sub-query

- Sub-queries return a single constant, this constant can be compared with another value in a WHERE clause
- Sub-queries return relations, that can be used in WHERE clause
- Sub-queries can appear in FROM clauses, followed by a tuple variable

Sub-queries that Produce Scalar Values

An atomic value that can appear as one component of a tuple is referred to as a scalar

Let’s compare two queries for the same request

Sub-queries that Produce Scalar Values

Example 7: Find the employees of *Phòng Phần mềm trong nước* department

```
/*7*/
SELECT *
FROM tblEmployee E, tblDepartment D
WHERE e.depNum=d.depNum AND d.depName LIKE N'Phòng phần mềm trong nước';
GO
```

Sub-queries that Produce Scalar Values

Example 11:

- Find the employees of *Phòng Phần mềm trong nước* department

```
/*11*/
SELECT *
FROM tblEmployee
WHERE depNum = (SELECT depNum
                FROM tblDepartment
                WHERE depName=N'Phòng Phần mềm trong nước')
GO
```

Sub-queries that Produce Scalar Values

Example 11:

- Find the employees of *Phòng Phần mềm trong nước* department

```
SELECT   *
FROM     tblEmployee
WHERE    depNum =ANY (SELECT depNum
                      FROM tblDepartment
                      WHERE depName=N'Phòng Phần mềm trong nước')
GO
```

Sub-queries that Produce Scalar Values

Example 11:

- Find the employees of *Phòng Phần mềm trong nước* department

```
SELECT   *
FROM     tblEmployee
WHERE    depNum IN (SELECT depNum
                    FROM tblDepartment
                    WHERE depName=N'Phòng Phần mềm trong nước')
GO
```

Conditions Involving Relations

Some SQL operators can be applied to a relation R and produce a bool result

- (EXISTS R = True) $\Leftrightarrow$ R is not empty
- (s IN R = True) $\Leftrightarrow$ s is equal to one of the values of R
- (s > ALL R = True) $\Leftrightarrow$ s is greater than every values in unary R
- (s > ANY R = True) $\Leftrightarrow$ s is greater than at least one value in unary R

Conditions Involving Tuples

A tuple in SQL is represented by a list of scalar values between ()

If a tuple t has the same number of components as a relation R, then we may compare t and R with IN, ANY, ALL

Sub-queries that Produce Scalar Values

Example 12:

- Find the dependents of all employees of department number 1

```
/*12*/
SELECT *
FROM tblDependent
WHERE empSSN IN (SELECT empSSN
                 FROM tblEmployee
                 WHERE depNum=1)
GO
```

Correlated Sub-queries

To now, sub-queries can be evaluated once and for all, the result used in a higher-level query

But, some sub-queries are required to be evaluated many times

That kind of sub-queries is called correlated sub-query

Note: *Scoping rules* for names

Correlated Sub-queries

Example 13:

- Find all those projects have the same location with projectA

```
/*13*/
SELECT * FROM tblProject
WHERE locNum = (SELECT p.locNum
                FROM tblProject p
                WHERE p.proName=N'ProjectA')
GO
```

Correlated Sub-queries

Another example:

- Find the titles that have been used for two or movies

```
SELECT title
FROM Movies Old
WHERE year < ANY
  (SELECT year
  FROM Movies
  WHERE title =Old.title)
```

Sub-queries in FROM Clauses

In a FROM list we can use a parenthesized sub-query

We must give it a tuple-variable alias

Example: Find the employees of *Phòng Phần mềm trong nước*

```
SELECT      *
FROM        tblEmployee e,
            (SELECT depNum
            FROM tblDepartment
            WHERE depName=N'Phòng phần mềm trong nước') d
WHERE       e.depNum=d.depNum
```

SQL Join Expressions

SQL Join Expressions can be stand as a query itself or can be used as sub-queries in FROM clauses

Cross Join in SQL= Cartesian Product

- Syntax: R CROSS JOIN S;
- Meaning: Each tuple of R connects to each tuple of S

Theta Join with ON keyword

- Systax: R JOIN S ON R.A=S.A;
- Meaning: Each tuple of R connects to those tuples of S, which satisfy the condition after ON keyword

SQL Join Expression

Example 15.1

- Product two relations Department and Employee

Example 15.2

- Find departments and employees who work in those departments, respectively

```
SELECT *
FROM tblDepartment d JOIN tblEmployee e ON d.depNum=e.depNum
GO
```

SQL Join Expression

More Example

Natural Joins

A natural join differs from a theta-join in that:

- The join condition: all pairs of attributes from the two relations having a common name are equated, and there are no other condition
- One of each pair of equated attributes is projected out
- Syntax : Table1 NATURAL JOIN Table2
- **Microsoft SQL SERVER DONOT SUPPORT NATURAL JOINS AT ALL**

Outer Joins

The outer join is a way to augment the result of join by the dangling tuples, padded with null values

When padding dangling tuples from both of its arguments, we use *full outer join*

When padding from left/right side, we use *left outer join/right outer join*

Outer joins

Example 17.1:

- For each location, listing the projects that are processed in it

```
/*17.1*/
SELECT l.locNum,l.locName,p.proNum,p.proName
FROM tblLocation l LEFT OUTER JOIN tblProject p ON l.locNum=p.locNum;
GO
```

Outer joins

Example 17.2:

- For each department, listing the projects that it controls

```
/*17.2*/
SELECT d.depName,p.proName
FROM tblDepartment d LEFT OUTER JOIN tblProject p ON d.depNum=p.depNum
GO
```

6.4 Full-Relation Operations

Study some operations that acts on relations as whole, rather than on tuples individually

Eliminating Duplicates

A relation, being a set, cannot have more than one copy of any given tuple

But, the SQL response to a query may list the same tuple several times, that is, SELECT preserves duplicates as a default

So, by DISTINCT we can eliminate a duplicates from SQL relations

Eliminating Duplicates

Example 17.3: List all location in which the projects are processed.

- Location name is repeated many times

```
SELECT DISTINCT l.locNum, l.locName
FROM tblLocation l JOIN tblProject p ON l.locNum=p.locNum
```

```
SELECT DISTINCT l.locNum, l.locName
FROM tblLocation l JOIN tblProject p ON l.locNum=p.locNum
```

Duplicates in Unions, Intersections, and Differences

Set operations on relations will eliminate duplicates automatically

Use ALL keyword after Union, Intersect, and Except to prevent elimination of duplicates

Syntax:

R UNION *ALL* S;

R INTERSECT *ALL* S;

R EXCEPT *ALL* S;

Grouping and Aggregation in SQL

Grouping operator partitions the tuples of relation into *groups*, based on the values of tuples in one or more attributes

After grouping the tuples of relation, we are able to *aggregate* certain other columns of relation

We use **GROUP BY** clause in SELECT statement

Aggregation Operators

Five aggregation operators

- SUM acts on single numeric column
- AVG acts on single numeric column
- MIN acts on single numeric column
- MAX acts on single numeric column
- COUNT act on one or more columns or all of columns

Eliminating duplicates from the column before applying the aggregation by DISTINCT keyword

Aggregation Operators

Example 18.1

- Find average salary of all employees

Example 18.2

- Find number of employees

```
/*18.1*/
SELECT AVG(empSalary) AS Average_Of_Salary
FROM tblEmployee
GO

/*18.2*/
SELECT COUNT(*) AS Count_Of_Employees
FROM tblEmployee
GO
```

Grouping

To partition the tuples of relation into groups

Syntax

SELECT <list of attributes>

FROM <list of tables>

WHERE <condition>

GROUP BY <list of attributes>

Grouping

Example 19.1:

- Group employees by department number

Example 19.2

- List number of employees for each department number

```
/*19.1*/
SELECT *
FROM tblEmployee
ORDER BY depNum
GO
```

```
/*19.2*/
SELECT depNum, COUNT(*) AS Num_Of_Employees
FROM tblEmployee
GROUP BY depNum
ORDER BY count(*) ASC
GO
```

Grouping

There are two kinds of terms in SELECT clause

- *Aggregations*, that applied to an attribute or expression involving attributes
- *Grouping Attributes*, that appear in GROUP BY clause

A query with GROUP BY is interpreted as follow:

- Evaluate the relation R expressed by the FROM and WHERE clauses
- Group the tuples of R according to the attributes in GROUP BY clause
- Produce as a result the attributes and aggregation of the SELECT clause

Grouping

Example 20

- Compute the number of employees for each project

```
/*20*/
SELECT proNum,COUNT(*) AS Num_Of_Employees
FROM tblWorksOn
GROUP BY proNum
GO
```

Grouping, Aggregation, and Nulls

When tuples have nulls, there are some rules:

- The value NULL is ignored in any aggregation
  - Count(*): a number of tuples in a relation
  - Count(A): a number of tuples with non-NULL values for A attribute
- NULL is treated as an ordinary value when forming groups
- The count of empty bag is 0, other aggregation of empty bag is NULL

Grouping, Aggregation, and Nulls

Example: Suppose R(A,B) as followed

A	B
NULL	NULL

The result of query

```
SELECT A, count(B)
FROM R
GROUP BY A;
```

is one tuple (NULL,0)

The result of query

```
SELECT A, sum(B)
FROM R
GROUP BY A;
```

is one tuple (NULL,NULL)

Considerations …

If we want to apply conditions to tuples of relations, we put those conditions in WHERE clause

If we want to apply conditions to groups of tuples after grouping, those conditions are based on some aggregations, how can we do?

In that case, we follow the GROUP BY clause with a HAVING clause

HAVING clause

Syntax:

SELECT <list of attributes>

FROM <list of tables>

WHERE <conditions on tuples>

GROUP BY <list of attributes>

HAVING <conditions on groups>

HAVING clause

Example 21:

- Print the number of employees for each those department, whose average salary exceeds 80000

```
/*21*/
SELECT depNum, AVG(empSalary) AS Average_Of_Salary
FROM tblEmployee
GROUP BY depNum
HAVING AVG(empSalary)>80000
GO
```

HAVING clause

Some rules about HAVING clause

- An aggregation in a HAVING clause applies only to the tuples of the group being tested
- Any attribute of relations in the FROM clause may be aggregated in the HAVING clause, but only those attributes that are in the GROUP BY list may appear un-aggregated in the HAVING clause (the same rule as for the SELECT clause)

HAVING clause

Example:

```
SELECT proNum, COUNT(empSSN) AS Number_Of_Employees,
FROM  tblWorksOn
GROUP BY proNum
HAVING AVG(workHours)>20
```

```
SELECT proNum, COUNT(empSSN) AS Number_Of_Employees,
FROM  tblWorksOn
GROUP BY proNum
HAVING proNum=4
```