SLOT 9-13

CHAPTER 6

THE DATABASE LANGUAGE SQL

OBJECTIVES

Understand concepts of:

- 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

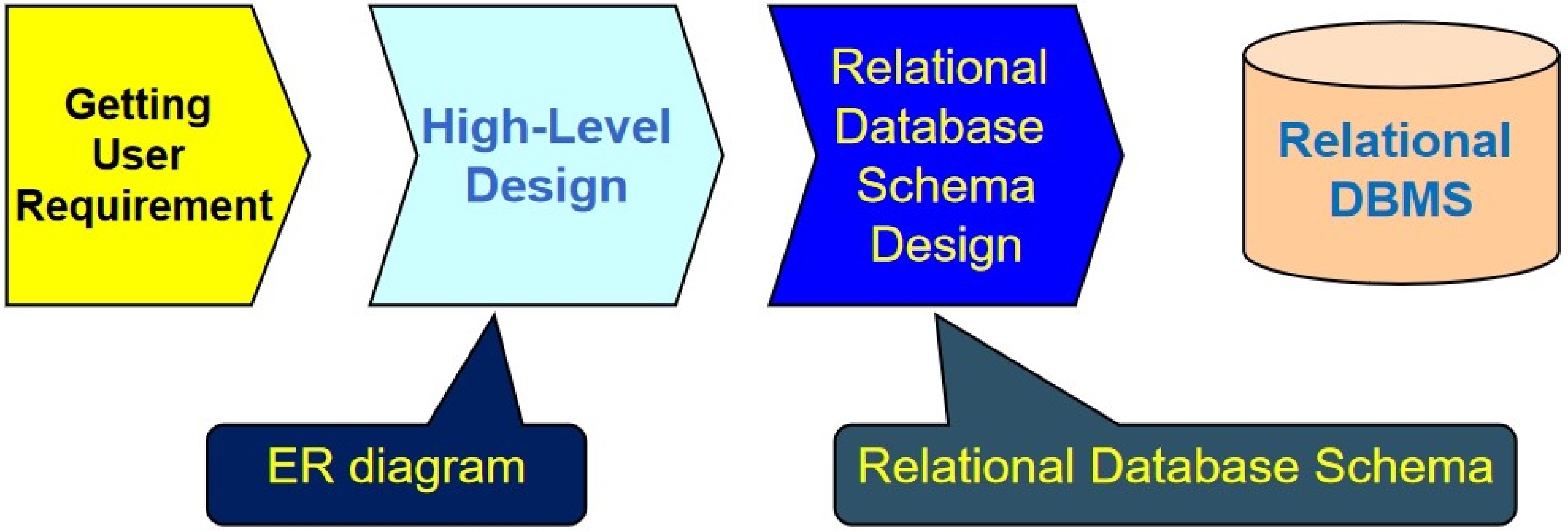
CONTENT

1) Integrity constraints (RB toàn vẹn)

2) Structure Query Language

- DDL (Data Definition Lanagauge)
- DML (Data Manipulation Language)
- DCL (Data Control Language) **(self studying)**
- Sub query

REVIEW

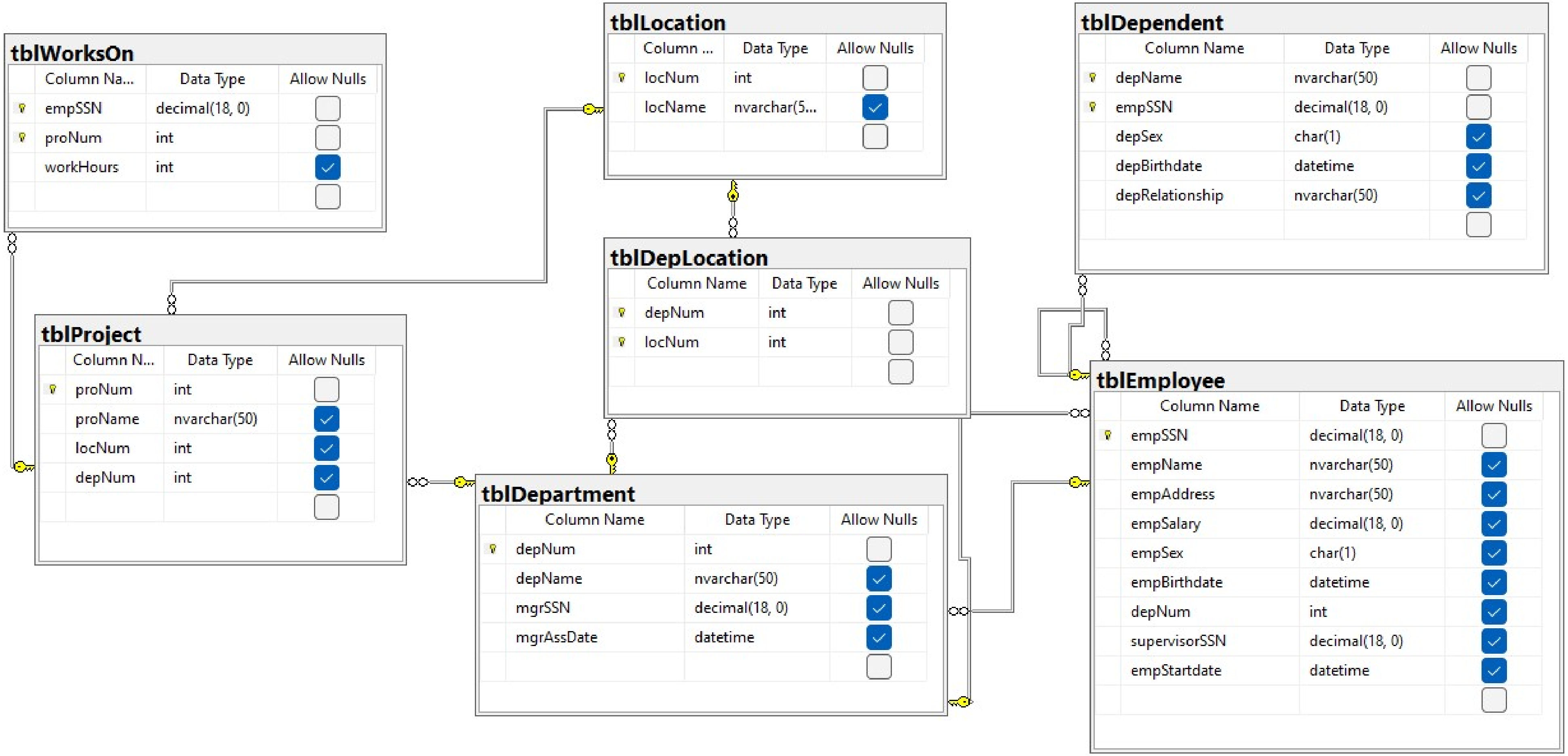


Studied:

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

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

THE COMPANY RELATIONAL DATABASE SCHEMA



tblWorksOn

	Column Na...	Data Type	Allow Nulls
🔑	empSSN	decimal(18, 0)	□
🔑	proNum	int	□
	workHours	int	☑
			□

tblLocation

	Column ...	Data Type	Allow Nulls
🔑	locNum	int	□
	locName	nvarchar(5...	☑
			□

tblDependent

	Column Name	Data Type	Allow Nulls
🔑	depName	nvarchar(50)	□
🔑	empSSN	decimal(18, 0)	□
	depSex	char(1)	☑
	depBirthdate	datetime	☑
	depRelationship	nvarchar(50)	☑
			□

tblDepLocation

	Column Name	Data Type	Allow Nulls
🔑	depNum	int	□
🔑	locNum	int	□
			□

tblProject

	Column N...	Data Type	Allow Nulls
🔑	proNum	int	□
	proName	nvarchar(50)	☑
	locNum	int	☑
	depNum	int	☑
			□

tblDepartment

	Column Name	Data Type	Allow Nulls
🔑	depNum	int	□
	depName	nvarchar(50)	☑
	mgrSSN	decimal(18, 0)	☑
	mgrAssDate	datetime	☑
			□

tblEmployee

	Column Name	Data Type	Allow Nulls
🔑	empSSN	decimal(18, 0)	□
	empName	nvarchar(50)	☑
	empAddress	nvarchar(50)	☑
	empSalary	decimal(18, 0)	☑
	empSex	char(1)	☑
	empBirthdate	datetime	☑
	depNum	int	☑
	supervisorSSN	decimal(18, 0)	☑
	empStartdate	datetime	☑
			□

1. INTEGRITY CONSTRAINTS

A "constraint" in a database refers to **a rule or condition** that limits the type of data that can be inserted into a table, ensuring data integrity and consistency

❑ **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

Type	Definition	Purpose	Enforcement
Domain integrity	Ensures that **all values in a column** fall within a defined set of **permissible values.**	To ensure that the data in each column is accurate and consistent.	Enforced through data types, constraints (CHECK), and rules (NOT NULL).
Entity integrity	Ensures that each table has a **primary key** and that the primary key values are **unique and not null**.	To ensure that each row in in a table can be uniquely identified.	Enforced through PRIMARY KEY constraints ensuring that no duplicate or null values exist in primary key columns.
Referential integrity	Ensures that a **foreign key value** always points to an **existing**, valid row in another table.	To maintain logical relationships between tables, preventing orphaned records.	Enforced through FOREIGN KEY constraints, ensuring that foreign key values match primary key values in related tables.

Data TYPE	USE CASE
INT	Used for integer values without decimals, suitable for counting, identifiers, and whole numbers.
DECIMAL	Used for exact numerical values with fixed precision and scale, suitable for financial calculations and quantities where exact precision is needed.
VARCHAR(n)	Used for variable-length character strings, suitable for text fields where the length can vary like names, emails, or descriptions.
TEXT	Used for large variable-length character strings, suitable for long text fields lik comments, articles, or product descriptions.
DATE	Used for date values, suitable for storing dates without time components like birthdays, anniversaries, or deadlines.
TIME	Used for time values, suitable for storing times without date components like office hours or appointment times.
DATETIME	Used for date and time values, suitable for storing precise moments in time like timestamps for events or logs.
BOOLEAN	Used for true/false values, suitable for binary conditions like status flags or feature toggles.

2. STRUCTURE QUERY LANGUAGE

2.1. DDL - **Data Definition Language**

2.2. DML - **Data Manipulation Language**

2.3. DCL (self studying)- **Data Control Language**

2.4. Sub query

SQL OVERVIEW

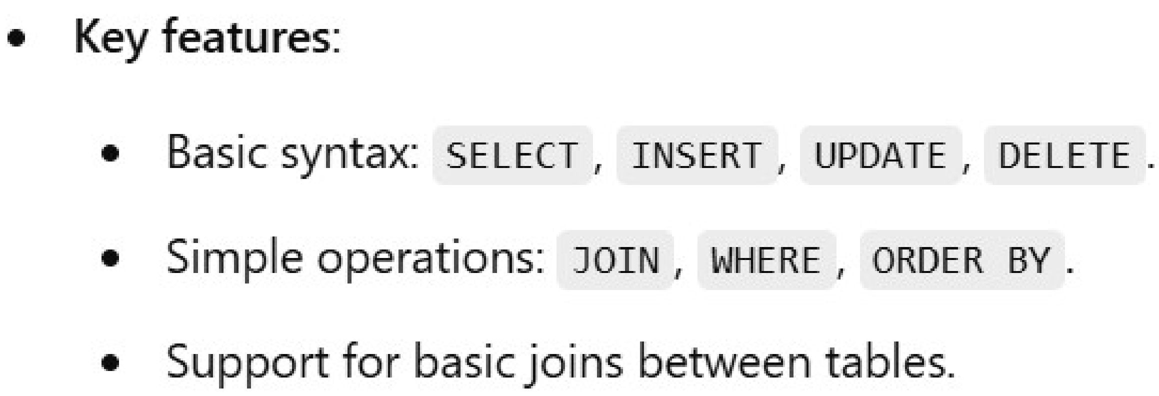
- SQL (sequel) is a database language designed for managing data in relational database management systems, and originally based upon relational algebra.
- **There are many different dialects of SQL**
  - Ansi SQL (or SQL-86), SQL-92, SQL-99
  - SQL:2003, SQL:2006, SQL:2008, SQL:2009
- **Transact-SQL** (**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

SQL OVERVIEW

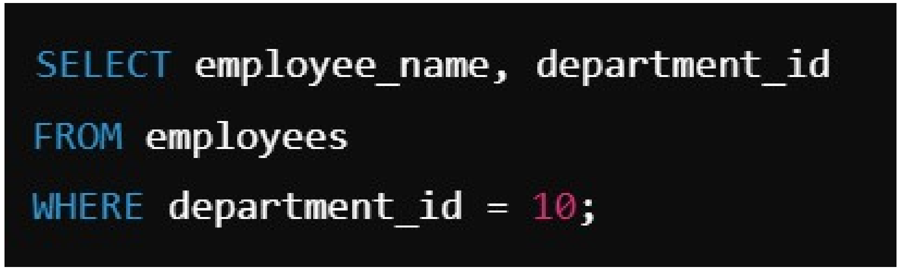
- SQL (sequel) is a database language designed for managing data in relational database management systems, and originally based upon relational algebra.
- **There are many different dialects of SQL**
  - Ansi SQL (or SQL-86), SQL-92, SQL-99
  - SQL:2003, SQL:2006, SQL:2008, SQL:2009
- **Transact-SQL** (**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

ANSI SQL (OR SQL-86)

SQL-86 (also known as SQL-1 or **ANSI SQL**) is the first version of SQL, standardized in 1986 by ANSI (American National Standards Institute).



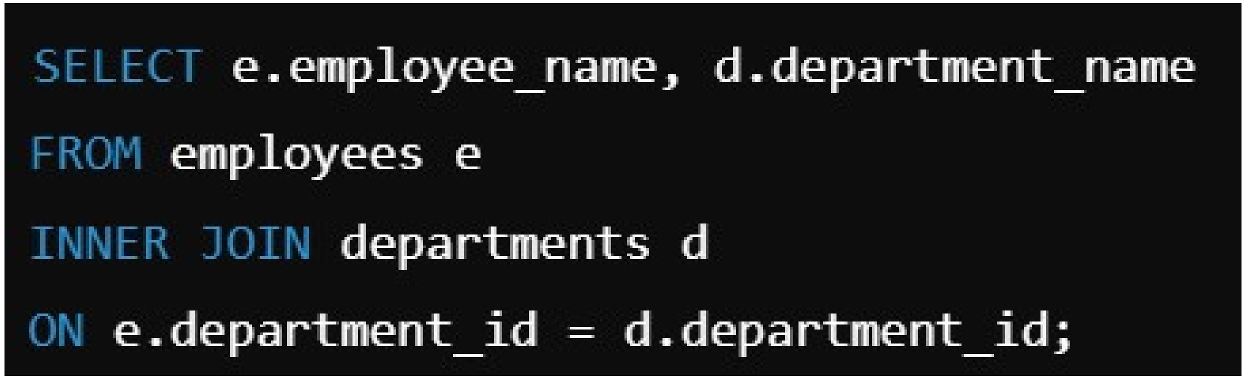
- Key features:
  - Basic syntax: `SELECT`, `INSERT`, `UPDATE`, `DELETE`.
  - Simple operations: `JOIN`, `WHERE`, `ORDER BY`.
  - Support for basic joins between tables.



```sql
SELECT employee_name, department_id
FROM employees
WHERE department_id = 10;
```

SQL-92

- New features:
  - Standardized `JOIN` syntax (INNER JOIN, LEFT JOIN, RIGHT JOIN).
  - `GROUP BY` and `HAVING` clauses to work with aggregate functions.
  - New data types like `DATE`, `TIME`, `TIMESTAMP`.
  - More complex subqueries.



```sql
SELECT e.employee_name, d.department_name
FROM employees e
INNER JOIN departments d
ON e.department_id = d.department_id;
```

DATA DEFINITION LANGUAGE - CREATE

- Database schema

  Simple syntax: **CREATE DATABASE** dbname

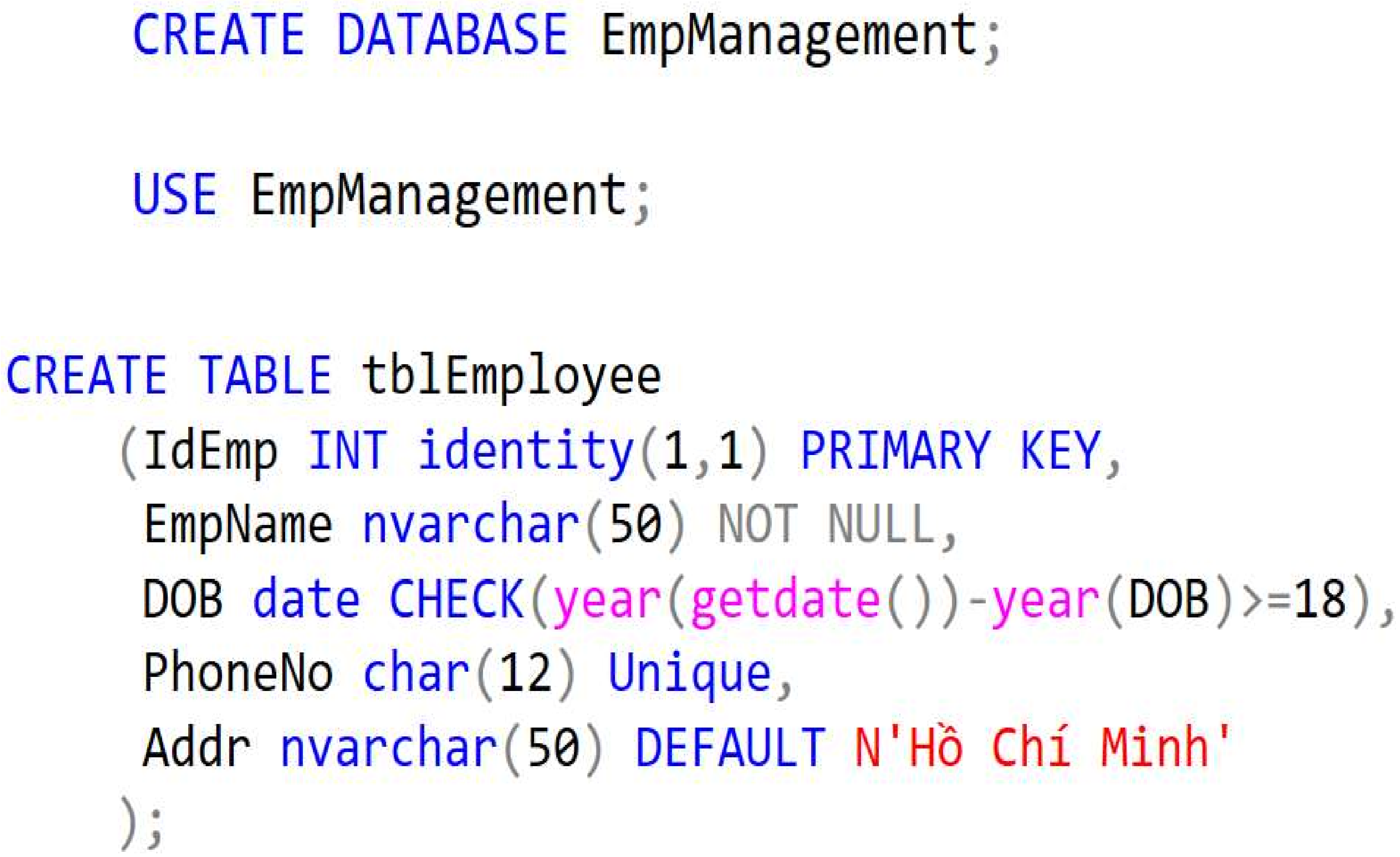
  Full syntax: https://docs.microsoft.com/en-us/sql/database

- Relation schema ~ table

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

```
CREATE TABLE tableName
    (
      fieldname1 datatype [integrity_constraints],
      fieldname2 datatype [integrity_constraints],
      ….
    )
```

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 column columnName
```

  - Modify data type

```
ALTER TABLE tableName
ALTER column columnName datatype [constraint]
```

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>);
```

Add/remove constraints

```
ALTER TABLE tablename
ADD CONSTRAINT constraintName CHECK
(expressionChecking)
```

(self studying)

```
ALTER TABLE tablename
DROP CONSTRAINT constraintName
```

Remove Table/ Database

```
DROP TABLE TableName
DROP DATABASE dbName
```

2.2. DML - Data Manipulation Language

01 **INSERT**

Click here to add text

02 **UPDATE**

Click here to add text

03 **DELETE**

Click here to add text

04 **SELECT**

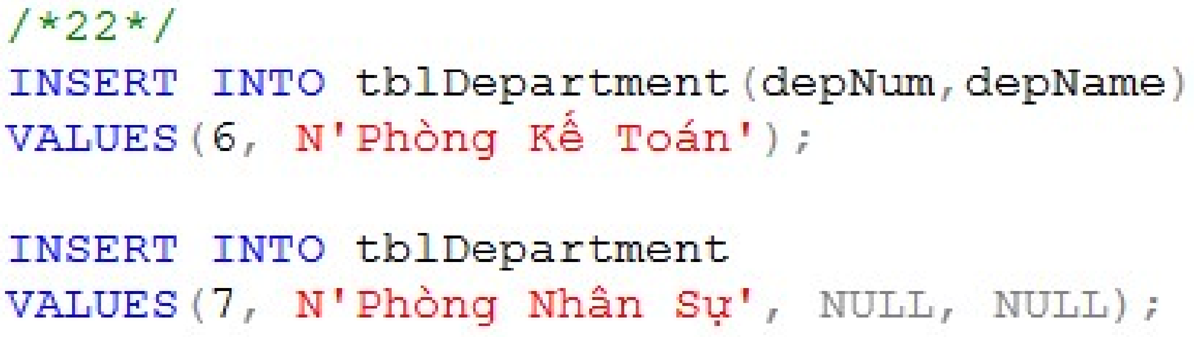
Click here to add text

1. INSERT

```
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);
```

2. UPDATE

UPDATE tableName

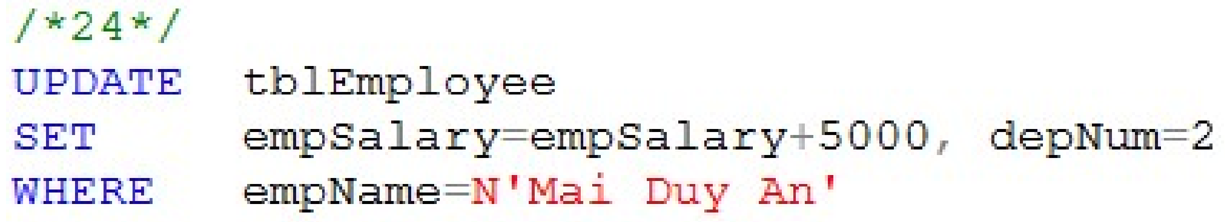
SET columnName = newValue

[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’



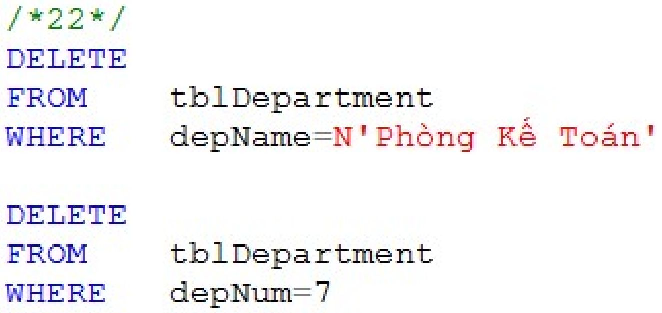
```
/*24*/
UPDATE  tblEmployee
SET     empSalary=empSalary+5000, depNum=2
WHERE   empName=N'Mai Duy An'
```

3. DELETE

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
```

4. SELECT

SQL Queries and Relational Algebra

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

$$\pi_L(\sigma_C(R))$$

4. SELECT

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

SELECT identifies *what* columns

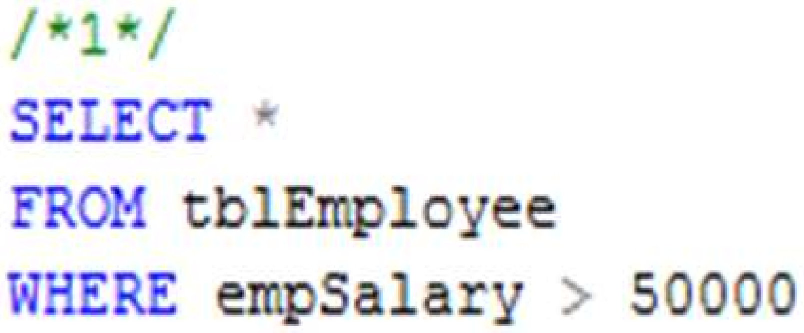
- ALL: Specifies that duplicate rows can appear in the result set. ALL is the default
- DISTINCT: Specifies that only unique rows can appear in the result set. Null values are considered equal for the purposes of the DISTINCT keyword
- TOP *n* [PERCENT]:Specifies that only the first *n* rows are to be 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

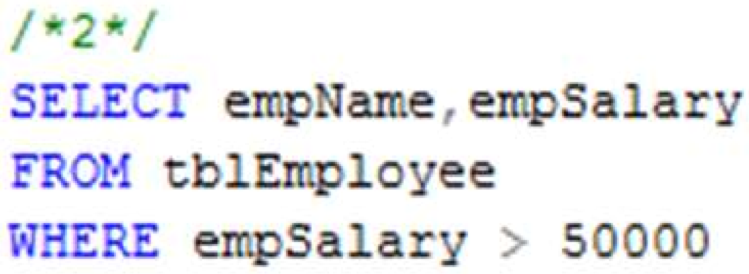
4. SELECT

Ex1: **Listing all employees whose salary exceed at 50000**



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

Ex2: **Listing name and salary of all employees whose income exceed 50000**



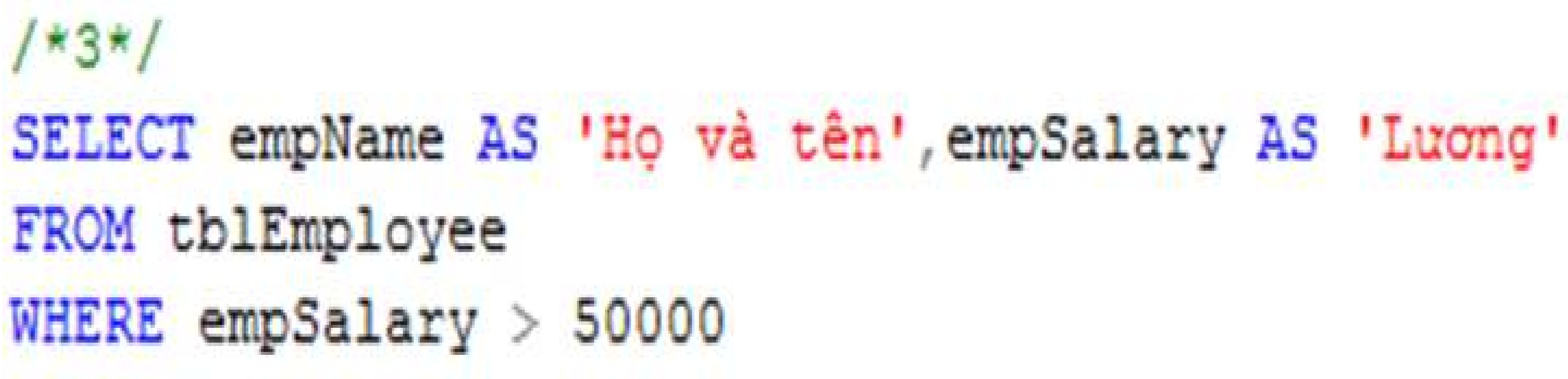
```
/*2*/
SELECT empName,empSalary
FROM tblEmployee
WHERE empSalary > 50000
```

4. SELECT

Using **alias** name in select clause

Example 3:

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

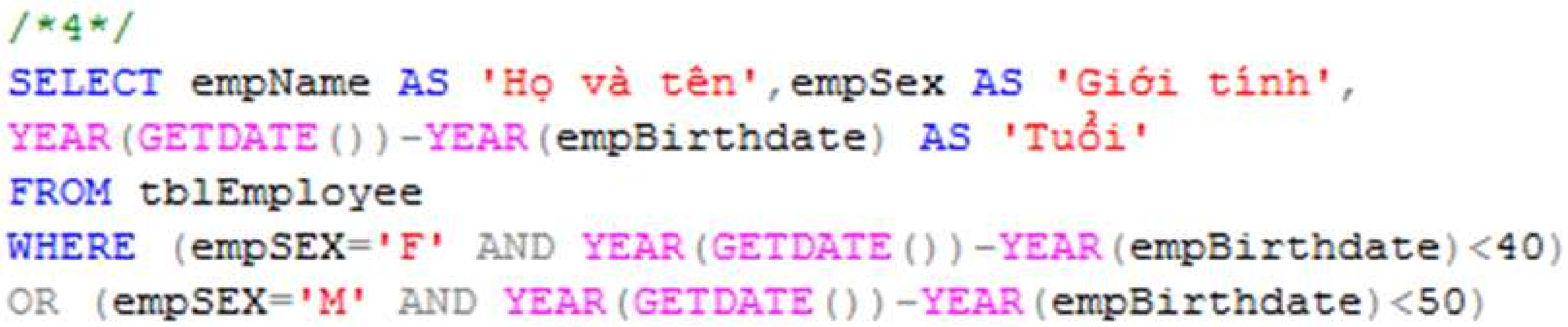


```
/*3*/
SELECT empName AS 'Họ và tên',empSalary AS 'Lương'
FROM tblEmployee
WHERE empSalary > 50000
```

4. SELECT

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)
```

4. SELECT - ORDER BY

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> [ASC/DESC]
```

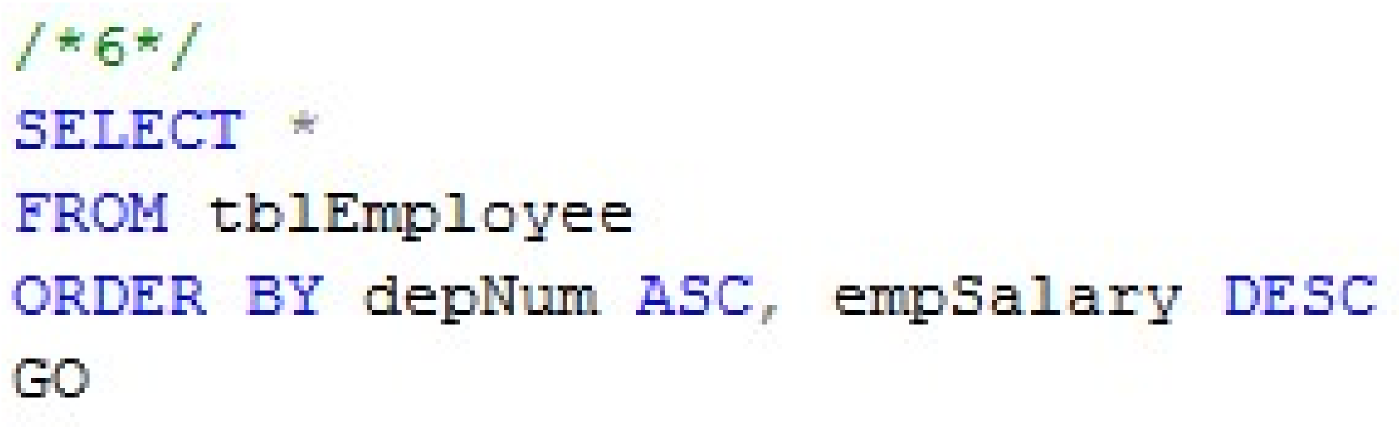
Order by clause follows Where and any other clauses. The ordering is performed on the result of the From, Where, and other clauses, just before Select clause

Using keyword **ASC** for ascending order and **DESC** for descending order

4. SELECT - ORDER BY

Example 6:

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



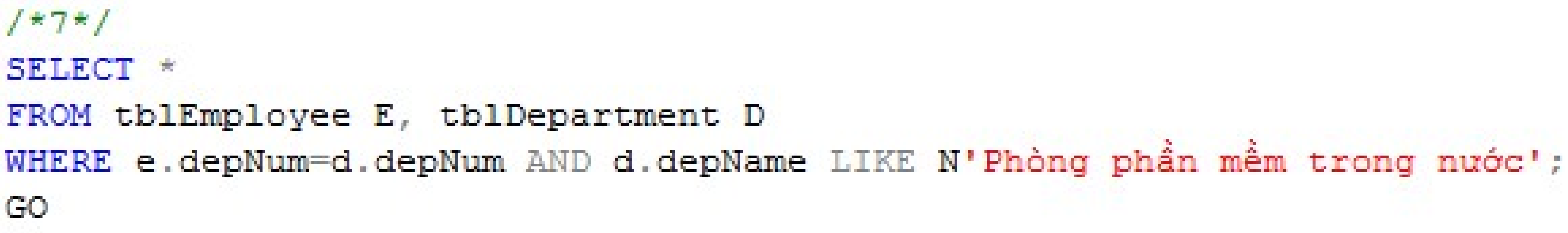
```
/*6*/
SELECT *
FROM tblEmployee
ORDER BY depNum ASC, empSalary DESC
GO
```

4. SELECT - JOINS

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

4. SELECT - JOINS

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
```

4. SELECT - JOINS

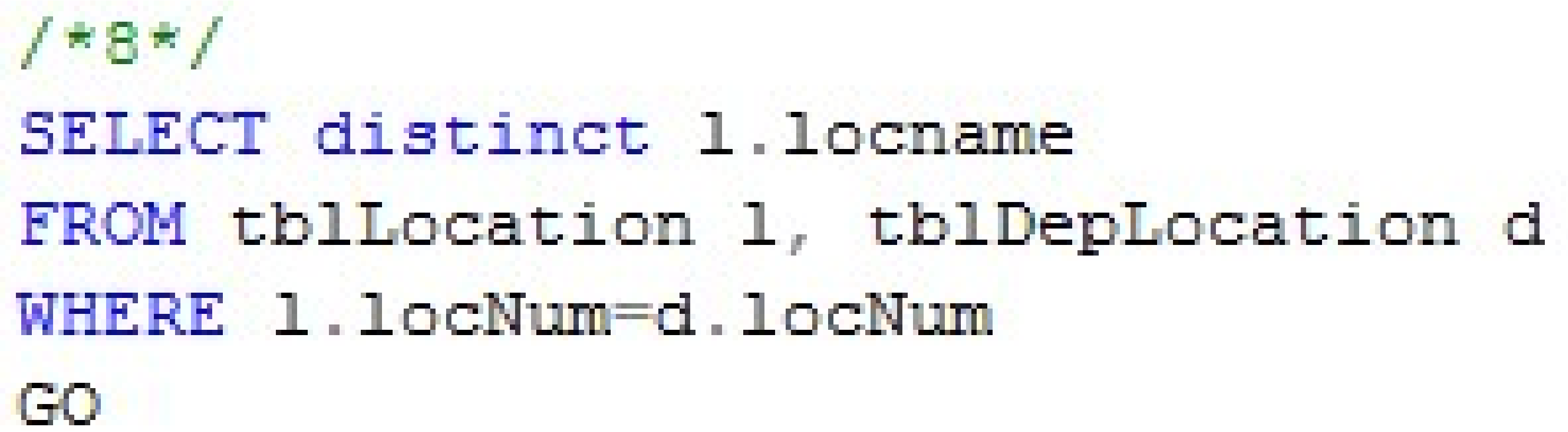
Questions:

- ...a query involves **several relations**, and there are two or more attributes with the same name?
- May we list a relation R as many times as we need?
- May we use tuple variables to refer to each occurrence of R?

4. SELECT - JOINS

Example 8:

Find all cities in which our company is



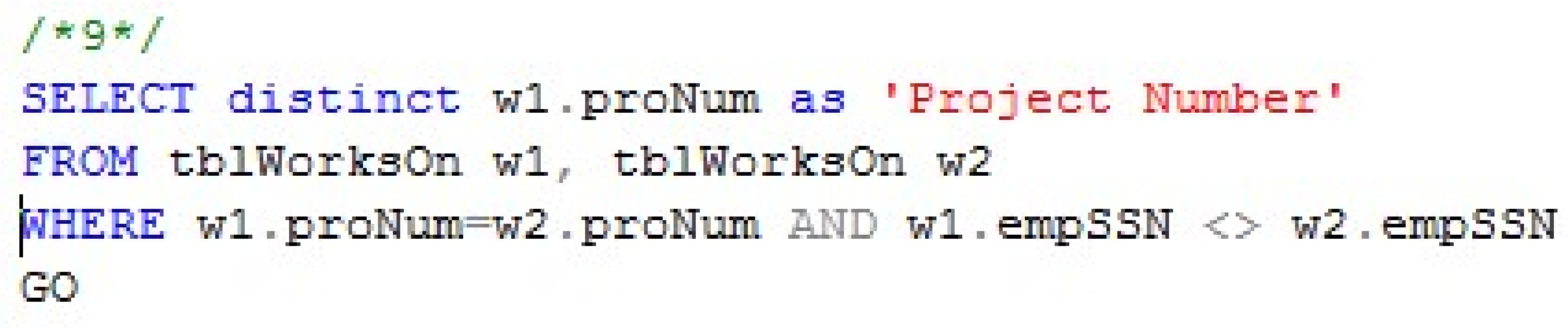
```
/*8*/
SELECT distinct l.locname
FROM tblLocation l, tblDepLocation d
WHERE l.locNum=d.locNum
GO
```

4. SELECT - JOINS

… 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
```

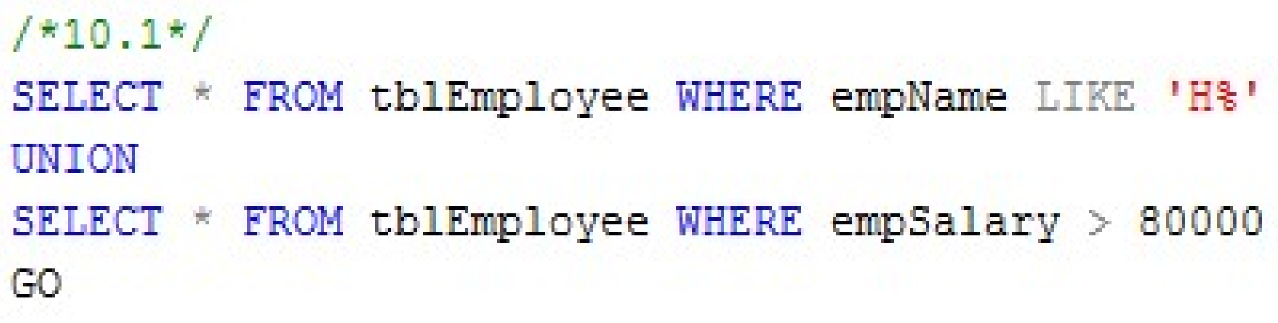
4. SELECT - **UNION**, **INTERSECT**, and **EXCEPT**

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

SQL provides corresponding operators with **UNION**, **INTERSECT**, and **EXCEPT** for ∪ , ∩, and -, respectively

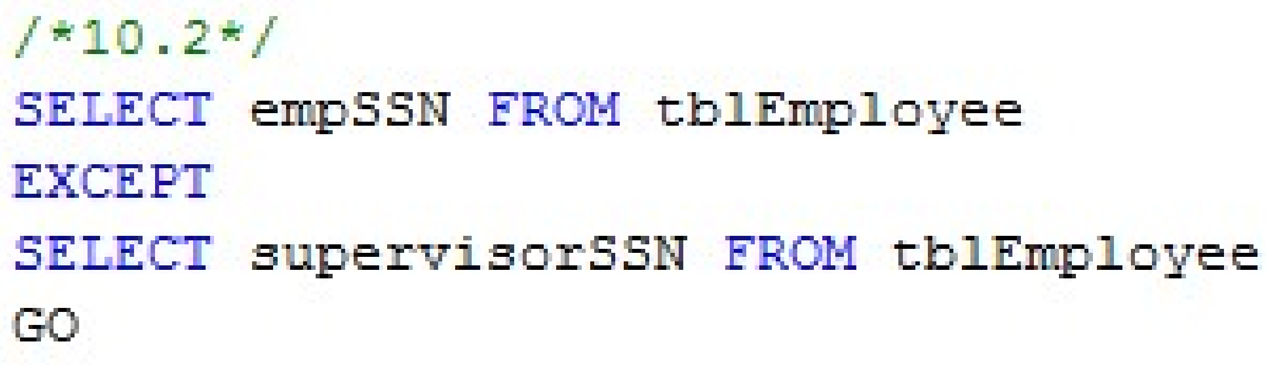
4. SELECT - UNION, INTERSECT, and EXCEPT

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
```

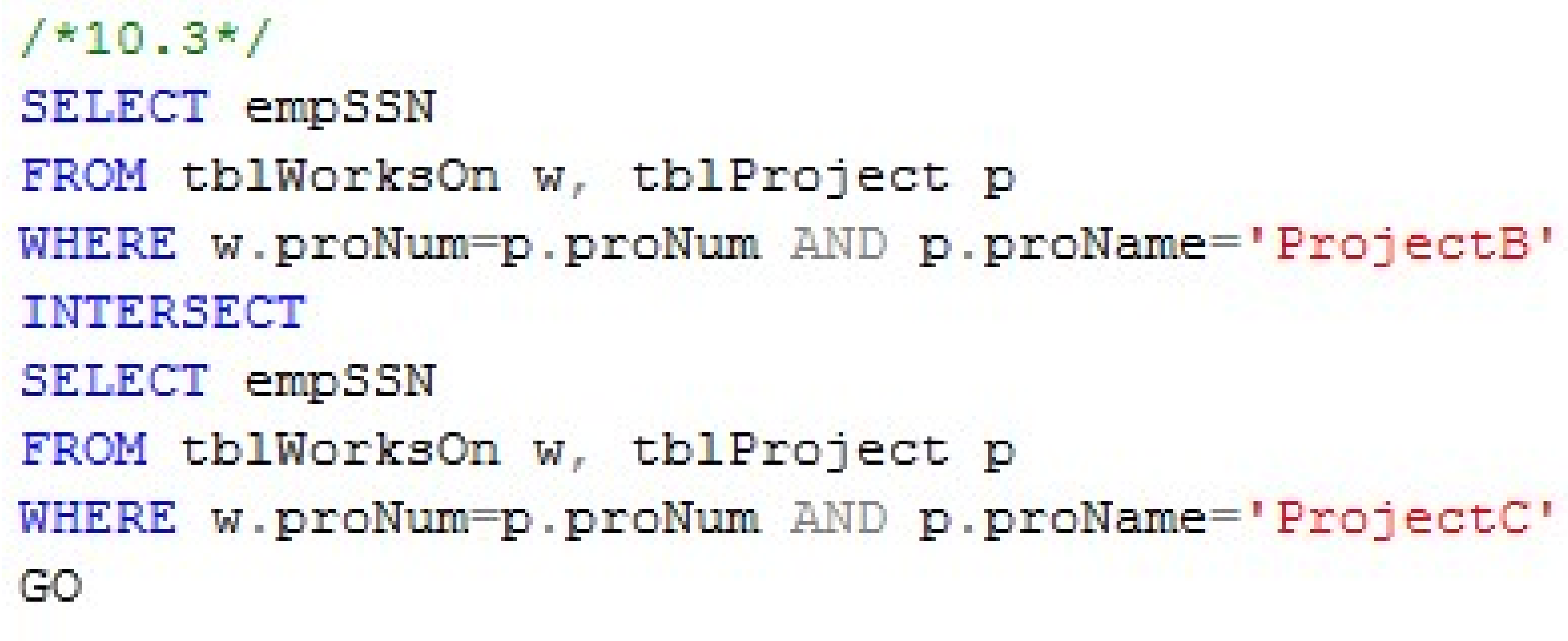
Example 10.2 Find all those *normal* employees, that is who do not supervise any other employees



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

Example 10.3

Find all employees who work on projectB and project C



```
/*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
```

4. SELECT - sub-query

- A query can be used to help in the evaluation of another

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

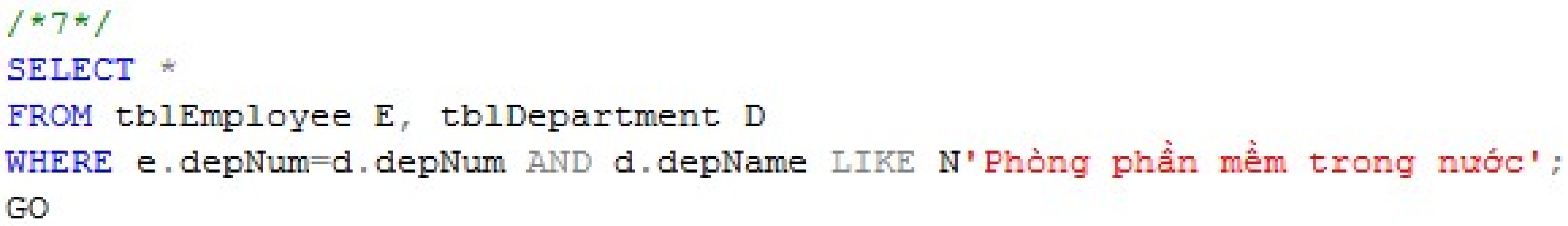
  - ❖ Sub-queries return a single constant, this constant can be compared with another value in a WHERE clause
  - ❖ Sub-queries return relations, that can be used in WHERE clause
  - ❖ Sub-queries can appear in FROM clauses, followed by a tuple variable

4. SELECT - sub-query

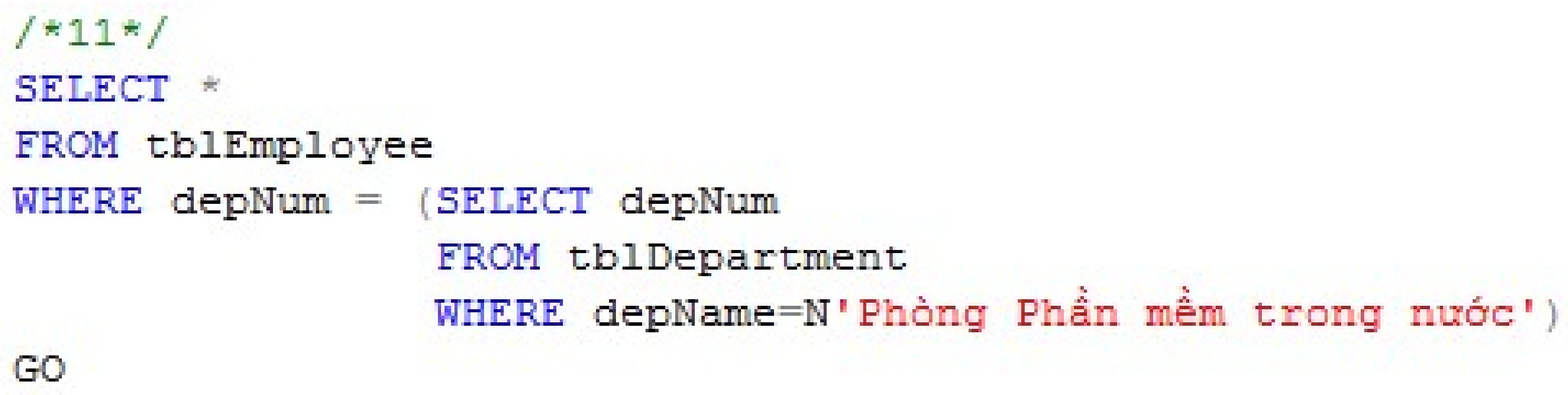
An atomic value that can appear as one component of a tuple is referred to as a **scalar.**

Let’s compare two queries for the same request

Example 7&11: 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
```



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

4. SELECT - EXIST-IN-ALL-ANY

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

(**EXISTS** R = True) ⇔ R is not empty

(s **IN** R = True) ⇔ S is equal to one of the values of R

(s > **ALL** R = True) ⇔ s is greater than every values in unary R

(s > **ANY** R = True) ⇔ s is greater than at least one value in unary R

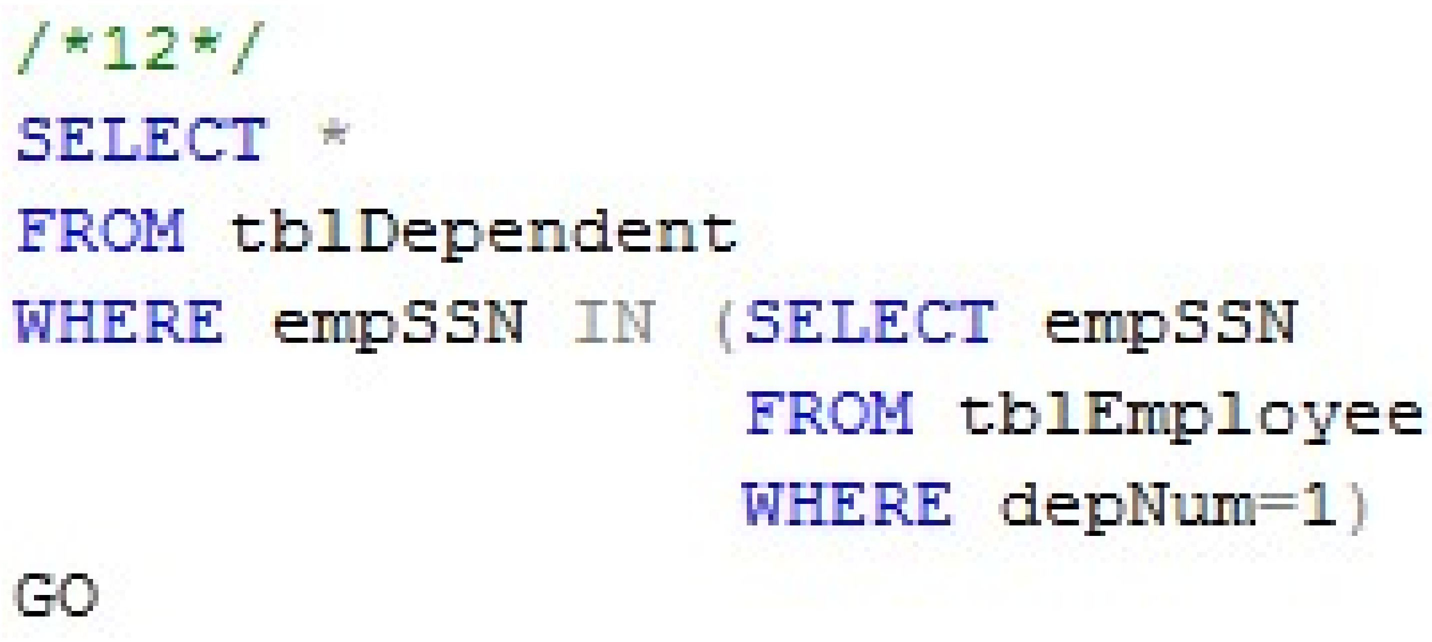
4. SELECT - BETWEEN...AND

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**

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
```

4. SELECT (CONT.)

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

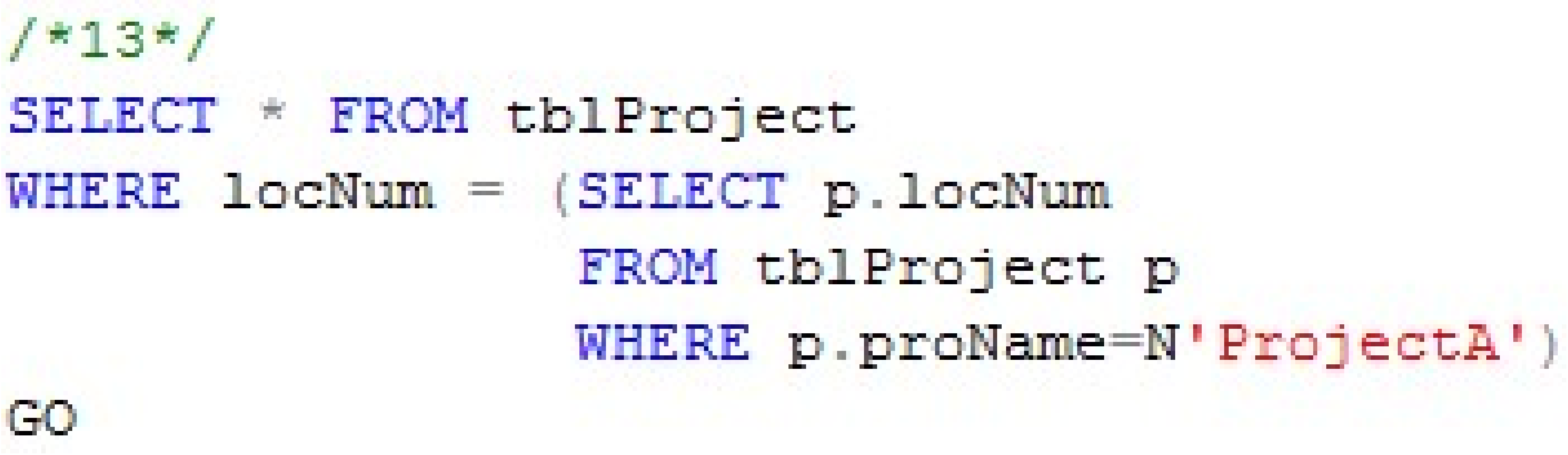
But, some sub-queries are required to be evaluated many times

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

Note: *Scoping rules* for names

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
```

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 QUERY

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
```

SUB QUERY

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

SUB QUERY

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;

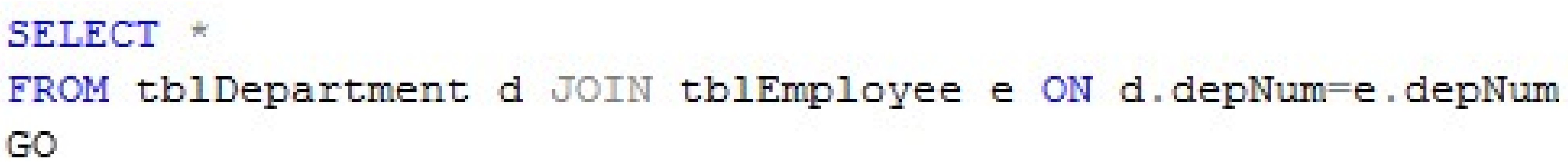
Meaning: Each tuple of R connects to those tuples of S, which satisfy the condition after ON keyword

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
```

NATURAL JOIN

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**

NATURAL JOIN

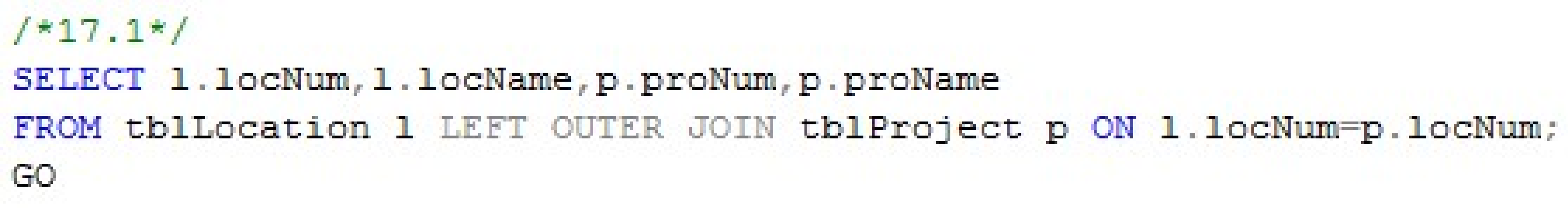
The outer join is a way to augment the result of join by the dangling tuples, padded with null values

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*

Example 17.1:

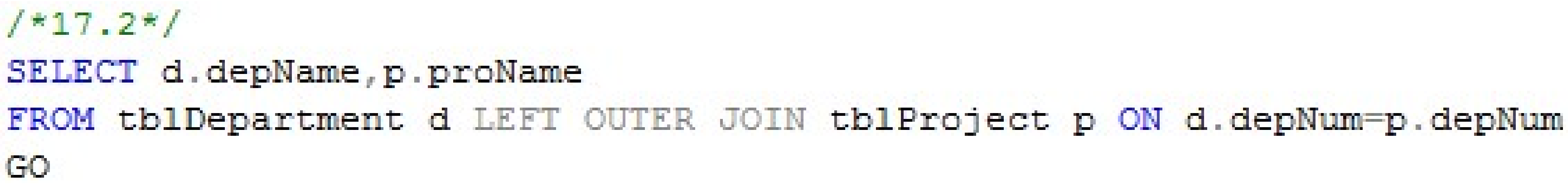
For each location, listing the projects that are processed in it



```sql
/*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
```

Example 17.2:

For each department, listing the projects that it controls



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

SELECT DISTINCT

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

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

SELECT DISTINCT

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
```

ALL

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;

GROUP BY

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

SUM - AVG - MIN - MAX - COUNT

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

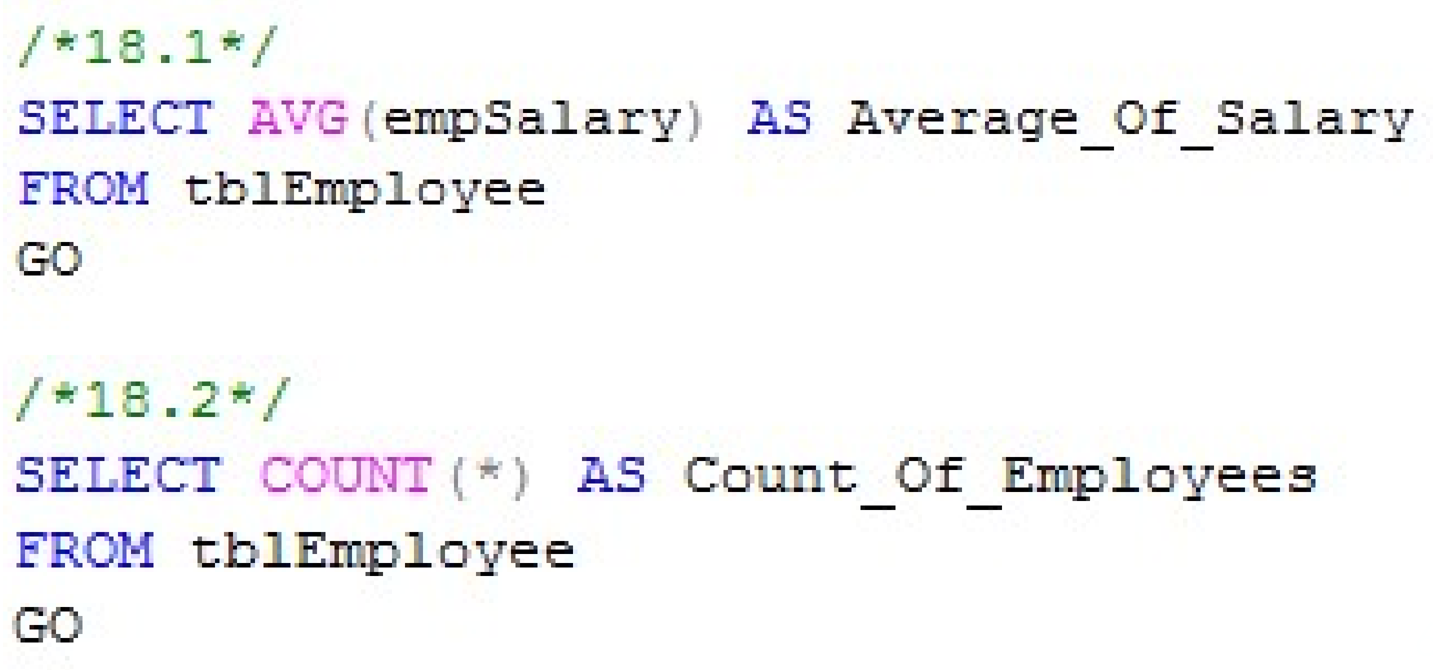
SUM - AVG - MIN - MAX - COUNT

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
```

SUM - AVG - MIN - MAX - COUNT

To partition the tuples of relation into groups

Syntax

```
SELECT <list of attributes>
FROM <list of tables>
WHERE <condition>
GROUP BY <list of attributes>
```

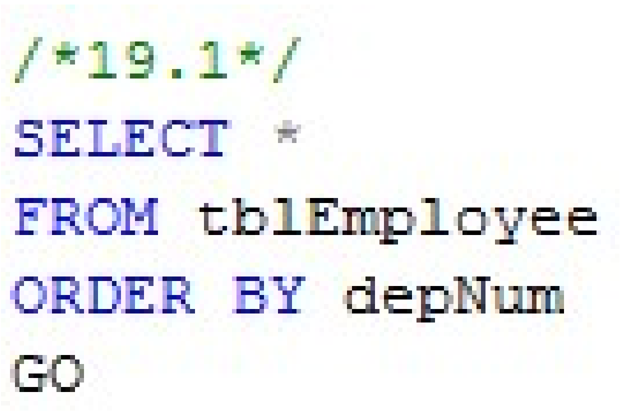
SUM - AVG - MIN - MAX - COUNT

Example 19.1:

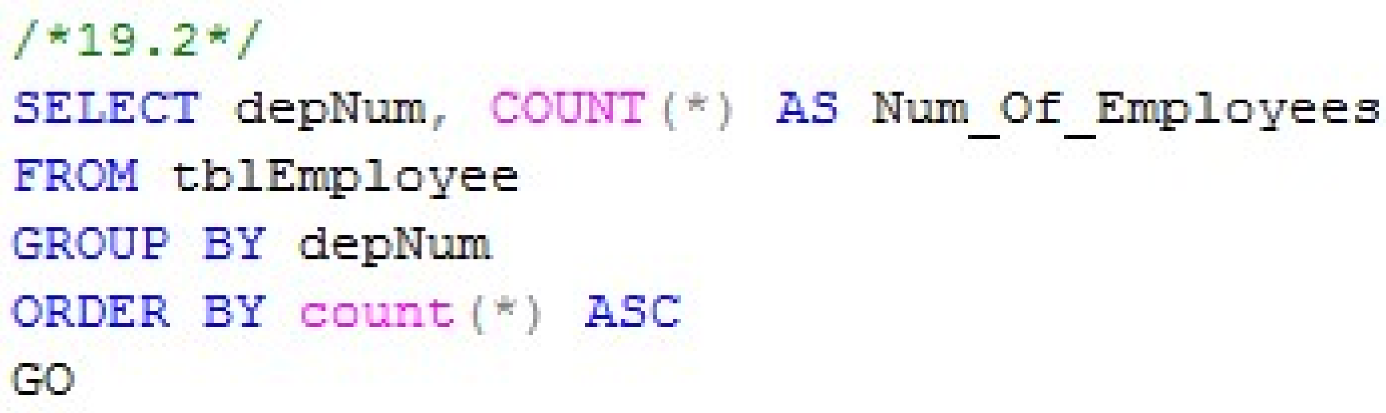
- Group employees by department number

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
```

SELECT

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

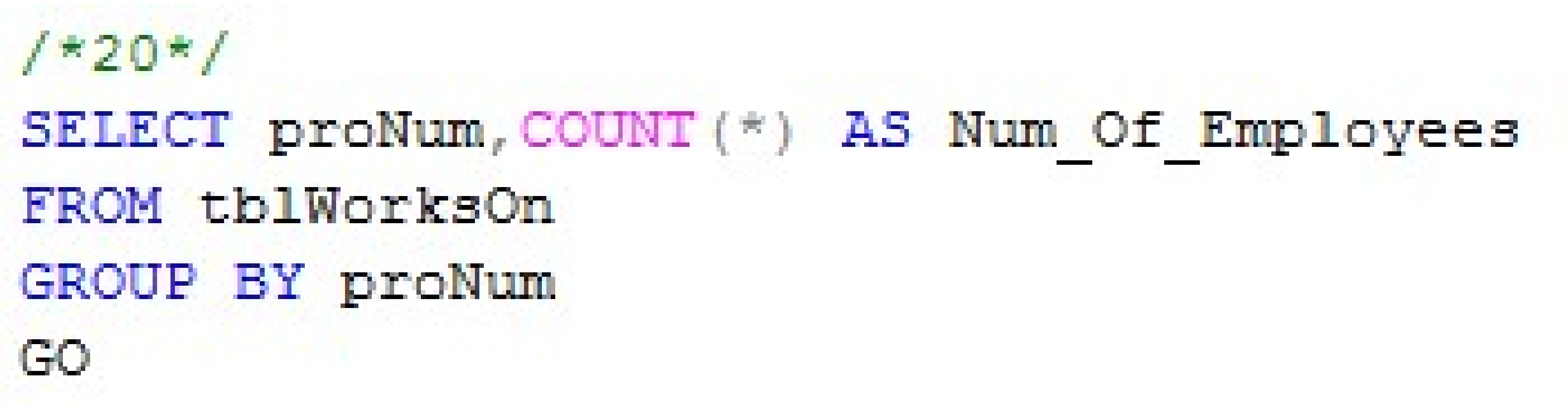
A query with GROUP BY is interpreted as follow:

- ❑ 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

SELECT

Example 20

- Compute the number of employees for each project



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

SELECT - 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

Example: Suppose R(A,B) as followed

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)

A	B
NULL	NULL

SELECT - HAVING

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

SELECT - HAVING

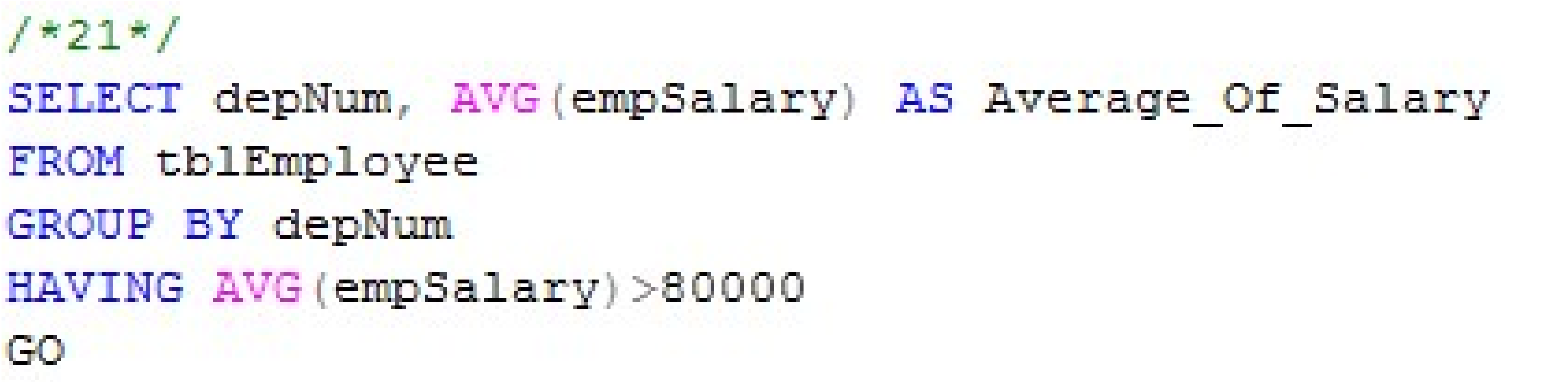
Syntax:

```
SELECT <list of attributes>
FROM  <list of tables>
WHERE  <conditions on tuples>
GROUP BY <list of attributes>
HAVING <conditions on groups>
```

SELECT - HAVING

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
```

SELECT - HAVING

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)

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
```

COMPARITION

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

$\exists\, k \le \min(n,m)$:

$\forall i,\ 1 \le i \le k$: $a_i = b_i$, and

$a_{k+1} < b_{k+1}$

Example

***fo**dder* < ***fo**o*

bar < ***bar**gain*

PATTERN MATCHING IN SQL

Like or Not Like

```
SELECT

FROM

WHERE s LIKE p;
```

```
SELECT

FROM

WHERE s NOT LIKE p;
```

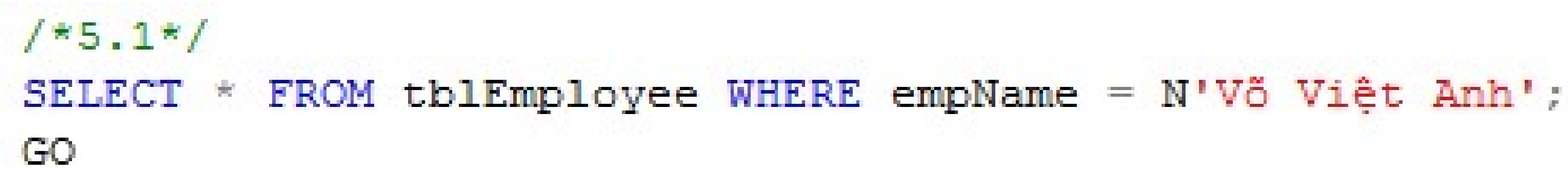
Two special characters

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

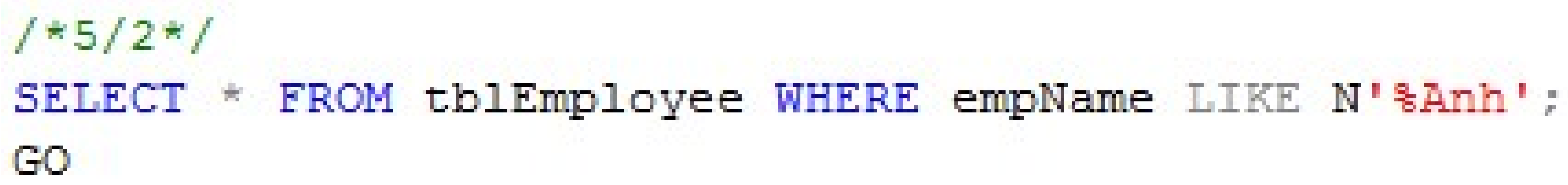
PATTERN MATCHING IN SQL

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