SQL: A COMMERCIAL DATABASE LANGUAGE

Basic Queries

Outline of this Chapter

1. Introduction
2. Data Definition, Basic Constraints, and Schema Changes
3. Basic Queries
4. More complex Queries
5. Aggregate Functions and Grouping
6. Summary of SQL queries
7. Data Change statements
8. Views
9. Complex Constraints
10. Database Programming

3. Basic Queries

- SQL has one basic statement for retrieving information from a database; the **SELECT statement.**

- Important distinction between SQL and the formal relational model: SQL allows a table (relation) to have two or more tuples that are *identical in all their attribute values*.
  Hence, an SQL relation (table) is a *multi-set* (sometimes called a *bag*) of tuples; it *is not* a set of tuples.
  SQL relations can be constrained to be sets by specifying PRIMARY KEY or UNIQUE attributes in a table definition, or by using the DISTINCT option in a query.

3.1. SELECT-FROM-WHERE BLOCK (1)

- Basic form of the SQL SELECT statement (also called *SELECT-FROM-WHERE block*):

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

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

- A SELECT-FROM-WHERE block is formed of the three clauses SELECT, FROM, WHERE.

3.1. SELECT-FROM-WHERE BLOCK (2)

- **Example:**
  The following query involves *one relation*.

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

SQL:

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

3.1. SELECT-FROM-WHERE BLOCK (3)

- **Example:**
  The following query involves *two relations*.

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

SQL:

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

– The WHERE-clause specifies the *selection condition* (corresponding to the selection operation in relational algebra) and a *join condition.*

3.1. SELECT-FROM-WHERE BLOCK (4)

- **Example:**

  The following query involves *three relations*.

  Query 2: Retrieve the phone number, department number, last name, birthdate and address of all employees who work for and manage departments that control projects at Stafford.

  SQL:

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

3.2. Dealing with ambiguous attribute names

- In SQL, we can use the same name for two (or more) attributes as long as the attributes are in *different relations*.

  A query that refers to two or more attributes with the same name must *qualify* the attribute name with the relation name by *prefixing* the relation name to the attribute name
  (e.g. DEPARTMENT.DNUMBER, DEPT_LOCATIONS.DNUMBER)

- Example:
  Query: list the department names and their locations.

```
SELECT   DEPARTMENT.DNAME,
         DEPT_LOCATIONS.DLOCATION
FROM     DEPARTMENT, DEPT_LOCATIONS
WHERE    DEPARTMENT.DNUMBER =
         DEPT_LOCATIONS.DNUMBER ;
```

3.3. Relation renaming (aliasing) (1)

- Some queries need to refer to the same relation twice.

  In this case, *aliases* are given to the *relation name*

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

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

- The alternate relation names E and S are called *aliases* for the EMPLOYEE relation.

- We can think of E and S as two *different copies* of the EMPLOYEE relation; E represents employees in the role of *supervisees* and S represents employees in the role of *supervisors.*

3.3. Relation renaming (aliasing) (2)

- Example (cont.):
  Alternative rewritings of Query 8 without the keyword **AS**:

```
SELECT E.FNAME, E.LNAME, S.FNAME,
       S.LNAME
FROM   EMPLOYEE E, EMPLOYEE S
WHERE  E.SUPERSSN = S.SSN ;
```

  And a more compact one:

```
SELECT E.FNAME, E.LNAME, S.FNAME,
       S.LNAME
FROM   EMPLOYEE E S
WHERE  E.SUPERSSN = S.SSN ;
```

3.3. Relation renaming (aliasing) (3)

- Aliasing can also be used in any SQL query for *convenience*.

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

```
SELECT  E.FNAME, E.LNAME, E.ADDRESS
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   D.DNAME='Research' AND
        D.DNUMBER = E.DNO ;
```

3.4. Attribute renaming (1)

- It is possible to *rename any attribute that appears in the result of a query* by adding the qualifier **AS** followed by the *desired new name*.

  Hence, the AS construct can be used to alias both *attribute* and *relation names*, and it can appear in both the SELECT and FROM clause of a query.

  Example:
  Query 8A: Retrieve the last name of each employee and the last name of her/his supervisor and rename the resulting attribute names as E_NAME and S_NAME respectively.

```
SELECT  E.LNAME AS E_NAME,
        S.LNAME AS S_NAME
FROM    EMPLOYEE AS E, EMPLOYEE AS S
WHERE   E.SUPERSSN= S.SSN ;
```

The answer of Query 8A:

E_NAME	S_NAME
Smith	Wang
...	...

3.4. Attribute renaming (2)

- It is also possible to *rename* the *relation attributes within the query in SQL* by giving them aliases.

Example:

In the FROM clause of a query we may write

EMPLOYEE **AS** E(FN, MI, LN, SSN, BD, ADDR, SEX, SAL, SSSN, DNO)

to rename EMPLOYEE relation and some of the attributes of the EMPLOYEE relation

3.5. Unspecified WHERE clause (1)

- A *missing WHERE-clause* indicates no condition; hence, *all tuples* of the relations in the FROM-clause are selected.

  This is equivalent to a TRUE condition in the WHERE clause.

  Example:
  Query 9: Retrieve the SSN values for all employees.

  SELECT SSN
  FROM EMPLOYEE

3.5. Unspecified WHERE clause (2)

- If more than one relation is specified in the FROM-clause *and* there is no join condition, then the *CARTESIAN PRODUCT* of tuples is selected.

  Example:
  Query 10: Retrieve all combinations of an EMPLOYEE SSN and a DEPARTMENT NAME.

  SELECT SSN, DNAME
  FROM EMPLOYEE, DEPARTMENT

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

3.6. Use of Asterisk (*) (1)

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

Examples:

Query 1C: Retrieve all the attribute values of the EMPLOYEE tuples for employees who work in department number 5.

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO=5;
```

Query 1D: Retrieve all the attributes of an EMPLOYEE and all the attributes of the DEPARTMENT he/she works in for very employee of the 'Research' department.

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

3.6. Use of Asterisk (*) (2)

Examples (cont.):

Query 10A: Compute the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

```
SELECT   *
FROM     EMPLOYEE, DEPARTMENT
```

3.7. Tables as sets – Use of DISTINCT (1)

- SQL *does not treat a relation as a set* (but as a multiset); *duplicate tuples can appear.*

- SQL *does not automatically eliminate tuples from the result of a query* because:
  - Duplicate elimination is expensive.
  - The user may want to see duplicates in the result of a query.
  - When an aggregate function is applied, in most cases we want to keep duplicates.

- Note that *a base table with a key is a set* since the key value cannot appear more than once in the table.

  Even though SQL does not require a table to have a key, in most cases there will be one.

3.7. Tables as sets – Use of DISTINCT (2)

- To eliminate duplicate tuples in a query result, the keyword **DISTINCT** in the SELECT clause is used.

- A query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL (which is equivalent to a simple SELECT) does not.

3.7. Tables as sets – Use of DISTINCT (3)

Example:

Query 11: Retrieve the salary of every employee.

```
SELECT ALL    SALARY
FROM          EMPLOYEE
```

Query 11A: Retrieve every distinct salary value of an employee.

```
SELECT DISTINCT   SALARY
FROM              EMPLOYEE
```

Answer of Q11

SALARY
30000
40000
25000
43000
38000
25000
25000
55000

Answer of Q11A

SALARY
30000
40000
25000
43000
38000
55000

3.8. Set operations (1)

- SQL has directly incorporated some set operations.

- There is a *union operation* **(UNION)**, a *set difference* **(EXCEPT)** and a *set intersection* **(INTERSECT)** operations.

- The resulting relations of these set operations are sets of tuples; *duplicate tuples are eliminated from the result.*

- The set operations apply only to *union compatible relations*: the two relations must have the same number of attributes and corresponding attributes must be of compatible data types.

3.8. Set operations (2)

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

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

3.9. Multiset operations

- SQL has also the corresponding multiset operation **UNION ALL**

The behavior of this operator is shown below.

R	A
	a1
	a2
	a2
	a3

S	A
	a1
	a2
	a4
	a5

R UNION ALL S

T	A
	a1
	a1
	a2
	a2
	a2
	a3
	a4
	a5

NOTE: MariaDB does not support EXCEPT ALL and INTERSECT ALL as they are confusing

3.10. Substring comparisons (1)

- The **LIKE** *comparison operator* is used to compare partial strings.

  Two reserved characters are used: '%' replaces an *arbitrary number of characters*, and '_' replaces *a single arbitrary character.*

Example**:**

Query 25: Retrieve all employee names whose address is in Houston, Texas. Here, the value of the ADDRESS attribute must contain the substring 'Houston,TX'.

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

3.10. Substring comparisons (2)

Example:

Query 26: Retrieve all employee names who were born during the 1950s.

Here, '5' must be the 9th character of the string (according to our format for date), so the BDATE value is '_ _ _ _ _ _ _ __5_', with each underscore as a placeholder for a single arbitrary character.

```
SELECT   FNAME, LNAME
FROM     EMPLOYEE
WHERE    BDATE LIKE  '_ _ _ _ _ _ _ __5_'
```

- The LIKE operator allows us to get around the fact that each value is considered atomic and indivisible; hence, *in SQL, character string attribute values are not atomic.*

3.11. Arithmetic and other operators (1)

- The standard arithmetic operators '+', '-'. '*', and '/' (for addition, subtraction, multiplication, and division, respectively) can be applied to numeric values or attributes with numeric domains.

Example:

Query 13: Show the effect of giving all employee salaries of employees who work on the 'ProductX' project a 10% raise.

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

3.11. Arithmetic and other operators (2)

- For string data types, the *concat function* can be used in a query to append two string values.

Example**:**

Query 27: Retrieve the names of employees in the form: first_name/blank_space/last_name.

SELECT **CONCAT(**FNAME, ‘ ‘, LNAME)
FROM EMPLOYEE

The answer of query 27:

CONCAT(FNAME, ‘ ‘, LNAME)
John Smith
Franklin Wong
…

3.11. Arithmetic and other operators (3)

- A comparison operator that can be used for *convenience* is **BETWEEN**.

  Example**:**
  Query 14: Retrieve all employees in department number 5 whose salary is between $30,000 and $40,000.

```
SELECT   *
FROM     EMPLOYEE
WHERE    DNO = 5 AND
      (SALARY BETWEEN 30000 AND 40000);
```

  The previous query is equivalent to:

```
SELECT   *
FROM     EMPLOYEE
WHERE    DNO = 5 AND
    SALARY >= 30000 AND SALARY <= 40000;
```

3.12. Query result ordering (1)

- The **ORDER BY** clause is used to sort the tuples in a query result based on the values of some attribute(s)

Example**:**

Query 15: Retrieve a list of employees and the projects each works in, ordered by the employee's department name, and within each department ordered alphabetically by employee last name, first name.

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

3.12. Query result ordering (2)

- The *default order* is in *ascending order* of values

We can specify the keyword **DESC** if we want a *descending order*.

The keyword **ASC** can be used to explicitly specify *ascending order*, even though it is the default.

Example:

Query 15A: Retrieve a list of employees and the projects each works in, ordered by the employee's department name in *descending order*, and within each department ordered alphabetically by employee last name, first name.

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

3.13. Explicit sets in SQL

- In SQL it is possible to use an *explicit (enumerated) set of values* in the WHERE clause of a query.

  Such a set is enclosed in parentheses and the keyword **IN** is used.

Example**:**

Query 17: Retrieve the social security number of employees who work on project number 1, 2 or 3.

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

3.13. NULLS in SQL

- NULLs in databases are used to represent a missing value.
  A NULL for an attribute A in a tuple t can have different meanings:
  - value unknown (we do not know if a value for A in t exists).
  - value exists but is not available.
  - Attribute A does not apply to tuple t.

  It is often not possible to determine which of the three meanings is intended.

- *SQL does not distinguish between the different meanings of NULL*.

- SQL considers *each NULL value is distinct from other NULL values*.

3.13. NULLS in SQL

- SQL allows queries that check *whether a value is* **NULL**.

  SQL uses **IS** or **IS NOT** (instead of using = and <>) to compare NULLs. Since in SQL each NULL value is distinct from other NULL values, *equality comparison is not appropriate*.

  Example**:**
  Query 14: Retrieve the names of all employees who do not have supervisors.

```
SELECT   FNAME, LNAME
FROM     EMPLOYEE
WHERE    SUPERSSN  IS  NULL;
```

3.14. NULLS and Three-Valued Logic (1)

- SQL uses a *3-valued logic with values TRUE (T), FALSE (F) and UNKNOWN (U)* instead of a the standard 2 valued logic with values TRUE and FALSE.

- When a NULL is involved in an *atomic comparison* in SQL, *the result is considered to be UNKNOWN*.

- The result of *logical expressions* that involve AND, OR and NOT are defined according to the following tables:

3.14. NULLS and Three-Valued Logic (2)

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT	
T	F
F	T
U	U

- What is the result of the logical expression:
  DNO = DNUMBER AND SALARY > 30000
  - when both DNO and SALARY are NULL ?
  - when only DNO is NULL ?

3.14. NULLS and Three-Valued Logic (3)

- Note: When a join condition is specified, tuples with null values for the join attributes *are not included in the result*.

  This rule does not apply to OUTER JOINs as we will see next.

3.15. Joined tables (1)

- The concept of a *joined table* allows users to specify a "joined relation" in the FROM-clause of a query.

  It looks like any other relation but is the result of a join

  It allows the user to specify *different types of joins* (regular "theta" JOIN, NATURAL JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, CROSS JOIN, etc)

3.15. Joined tables (2)

- Example**:**
  Query 1:

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

could be written as:

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

3.15. Joined tables (3)

- In a NATURAL JOIN *no JOIN condition is specified*. The implicit equi-join condition involves each pair of attributes with the *same name* in both relations.

- It is possible to *rename attributes* so that they *match* before applying NATURAL JOIN. This is achieved using an INNER QUERY as shown below. More on those in the next chapter.

Example**:**

Query 1 can be rewritten as follows:

```
SELECT  FNAME, LNAME, ADDRESS
FROM    (EMPLOYEE NATURAL JOIN
        (SELECT DNAME AS DNAME,
DNUMBER AS DNO, MGRSSN AS MSSN,
MGRSTARTDATE AS MSDATE FROM
DEPARTMENT)  AS DEPT
WHERE   DNAME='Research'
```

3.15. Joined tables (4)

Example of an OUTER JOIN:
Query 8:

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

If we want the names of employees that do not have a supervisor to be also included in the result we can write:

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

3.15. Joined tables (5)

- It is possible to *nest join specifications*.

  One of the tables in a join may itself be a joined table.

Example**:**

Query 2:

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