UNIT – 2

> **SQL – 1:** *SQL Data Definition and Data Types; Specifying basic constraints in SQL; Schema change statements in SQL; Basic queries in SQL; More complex SQL Queries.*

The name **SQL** is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R.

SQL is now the standard language for commercial relational DBMSs. A joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) has led to a standard version of SQL (ANSI 1986), called SQL-86 or SQL1.

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL *and* a DML.

SQL Data Definition and Data Types SQL uses the terms **table**, **row**, and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively.We will use the corresponding terms interchangeably.

The main SQL command for data definition is the **CREATE** statement,which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers).

Schema and Catalog Concepts in SQL Early versions of SQL did not include the concept of a relational database schema;

all tables (relations) were considered part of the same schema. An **SQL schema** is identified by a **schema name**, and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema.

Schema **elements** include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'.

Note that each statement in SQL ends with a semicolon. **CREATE SCHEMA** COMPANY **AUTHORIZATION** 'Jsmith';

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

The CREATE TABLE Command in SQL

The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL.

The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.

For example, by writing

CREATE TABLE *COMPANY.EMPLOYEE* (...............);
rather than
CREATE TABLE EMPLOYEE (EID VARCHAR(10),.......................); as shown above ,
we can explicitly (rather than implicitly) make the EMPLOYEE table part of the COMPANY schema.
The relations declared through CREATE TABLE statements are called **Base Tables** (or base relations).

This means that the relation and its tuples are actually created and stored as a file by the DBMS **Attribute Data Types and Domains in SQL .**

The basic **data types** available for attributes include **numeric, character string, bit string, Boolean, date, and time.**

 Numeric data types include integer numbers of various sizes (*INTEGER or INT, and SMALLINT*) and floating-point (real) numbers of various precision *(FLOAT or REAL, and DOUBLE PRECISION)*. Formatted numbers can be declared by using *DECIMAL(i,j)—or DEC(i,j) or NUMERIC(i,j).*

Character-string data types are either fixed length—*CHAR(n) or CHARACTER(n)*, where *n* is the number of characters—or varying length—*VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n*), where *n* is the maximum number of characters.

Another variable-length string data type called CHARACTER **LARGE OBJECT** or CLOB is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K),

megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.

Bit-string data types are either of fixed length n—BIT(n)—or varying length —BIT VARYING(n), where n is the maximum number of bits. The default for n, the length of a character string or bit string, is 1.

Boolean data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.

DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD.

The **TIME** data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form **HH:MM:SS**.

A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. example, TIMESTAMP '2008-09-27 09:12:47.648302'.

Another data type related to **DATE, TIME,** and **TIMESTAMP** is the **INTERVAL** data type.

This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp.

Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

Specifying Constraints in SQL These include *key and referential integrity constraints, restrictions on attribute domains and NULLs and constraints* on individual tuples within a relation.

Specifying Attribute Constraints and Attribute Defaults Because SQL allows NULLs as attribute values, a *constraint* NOT NULL may be specified if NULL is not permitted for a particular attribute.

It is also possible to define a *default value* for an attribute by appending the clause **DEFAULT** <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute.

Ex: **CREATE TABLE** EMPLOYEE (Ssn varchar(10),Ename varchar(10),salary integer,address varchar(10),....... Dno INT **NOT NULL DEFAULT** 1, **CONSTRAINT** EMPPK **PRIMARY KEY** (Ssn), **CONSTRAINT** EMPSUPERFK **FOREIGN KEY** (Super_ssn) **REFERENCES** EMPLOYEE(Ssn) **ON DELETE** SET NULL **ON UPDATE** CASCADE, **CONSTRAINT** EMPDEPTFK **FOREIGN KEY**(Dno) **REFERENCES** DEPARTMENT(Dnumber) **ON DELETE** SET DEFAULT **ON UPDATE** CASCADE);

Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition.

For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

*Dnumber INT **NOT NULL CHECK** (Dnumber > 0 **AND** Dnumber < 21);*

Specifying Key and Referential Integrity Constraints Because keys and referential integrity constraints are very important, there are special clauses within the CREATE TABLE statement to specify them.

For example, the primary key of DEPARTMENT can be specified as follows (instead of the way it is specified in Figure): *Dnumber INT **PRIMARY KEY**; We can specify RESTRICT, CASCADE, SET NULL or SET DEFAULT on referential integrity constraints (foreign keys)*

CREATE TABLE DEPT (DNAME VARCHAR(10) NOT NULL, DNUMBER INTEGER NOT NULL, MGRSSN CHAR(9), MGRSTARTDATE CHAR(9), PRIMARY KEY (DNUMBER), FOREIGN KEY (MGRSSN) REFERENCES EMP ON DELETE SET DEFAULT ON UPDATE CASCADE);

Specifying Constraints on Tuples Using CHECK :

other *table constraints* can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called **tuple-based** constraints because they apply to each tuple *individually* and are checked whenever a tuple is inserted or modified.

Ex: CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date. ***CHECK (Dept_create_date <= Mgr_start_date);***

Delete a Table or Database

To delete a table (the table structure, attributes, and indexes will also be deleted):

- **DROP TABLE** table_name

To delete a database:

- **DROP DATABASE** database_name

Truncate a Table

What if we only want to get rid of the data inside a table, and not the table itself? Use the **TRUNCATE TABLE** command (deletes only the data inside the table):

- **TRUNCATE TABLE** table_name

The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command.

For base tables, the possible *alter table actions* include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other elements) reference the column.

ALTER TABLE COMPANY.EMPLOYEE **DROP ADDRESS CASCADE;**

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

ALTER TABLE COMPANY.DEPARTMENT **ALTER MGRSSN DROP DEFAULT;**
ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN SET DEFAULT "333445555";

One can also change the constraints specified on a table by adding or dropping a constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 8.2 from the EMPLOYEE relation, we write:

ALTER TABLE **EMPLOYEE**
DROP CONSTRAINT EMPSUPERFK CASCADE;

The **ALTER TABLE** statement is used to add or drop columns in an existing table.

- **ALTER TABLE** table_name **ADD** column_name datatype
- **ALTER TABLE** table_name **DROP COLUMN** column_name
- **ALTER TABLE** table_name **MODIFY COLUMN** column_name
- **ALTER TABLE** table_name **RENAME** newtablename

Note: Some database systems don't allow the dropping of a column in a database table (DROP COLUMN column_name).

Person:

LastName	FirstName	Address
Kumar	Kiran	RR Nagar

Example
To add a column named "City" in the "Person" table:
ALTER TABLE Person **ADD** City varchar(30);
Result:

LastName	FirstName	Address	City
Kumar	Kiran	RR nagar	

Example
To drop the "Address" column in the "Person" table:
ALTER TABLE Person **DROP COLUMN** Address;
Result:

LastName	FirstName	City
Kumar	Kiran	Bangalore

The INSERT INTO Statement
The INSERT INTO statement is used to insert new rows into a table.

Syntax: **INSERT INTO** table_name **VALUES** (value1, value2,....)

We can also specify the columns for which you want to insert data:

INSERT INTO *table_name (column1, column2,...)***VALUES** *(value1, value2,....)*

Insert a New Row

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore

And this SQL statement:
INSERT INTO *Persons* ***VALUES*** *('Rao', 'Praveen', 'RajajiNagar', 'Bangalore')*
Will give this result:

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore

Insert Data in Specified Columns:

INSERT INTO *Persons* **(LastName, Address)VALUES (' Shetty ', ' banashankari ')**

Will give this result:

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty		**banashankari**	

The Update Statement
The UPDATE statement is used to modify the data in a table.
Syntax:

```
UPDATE table_name
SET column_name = new_value
WHERE column_name = some_value;
```

Person:

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty		banashankari	

Update one Column in a Row
We want to add a first name to the person with a last name of " Shetty ":

```
UPDATE Person SET FirstName = '  pradeep '
WHERE LastName = '  shetty ';
```

Result:

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty	Pradeep	banashankari	

Update several Columns in a Row
We want to change the address and add the name of the city:

```
UPDATE Person
SET  City = 'Tumkur'
WHERE LastName = 'shetty';
```

Result:

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore

Shetty	Pradeep	banashankari	Tumkur

The Delete Statement
The DELETE statement is used to delete rows in a table.
Syntax:

```
DELETE FROM table_name
WHERE column_name = some_value
```

Person:

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty	Pradeep	banashankari	Tumkur

Delete a Row
"Pradeep Shetty" is going to be deleted:

```
DELETE
FROM Person
WHERE LastName = 'shetty'
```

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore

Delete All Rows
It is possible to delete all rows in a table without deleting the table. This means that the tablestructure, attributes, and indexes will be intact:

DELETE FROM table_name;
or
DELETE * FROM table_name;

LastName	FirstName	Address	City

Try this: **SELECT * from Person;**

Examples:
The employee JONES is transfered to the department 20 as a manager and his salary is increased by 1000:

```
UPDATE EMP set
```

**JOB = 'MANAGER', DEPTNO = 20, SAL = SAL +1000
where ENAME = 'JONES';**

• All employees working in the departments 10 and 30 get a 15% salary increase.

**UPDATE EMP set
SAL = SAL * 1.15 where DEPTNO in (10,30);**

**UPDATE EMP set
SAL = (select min(SAL) from EMP
where JOB = 'MANAGER')
where JOB = 'SALESMAN' and DEPTNO = 20;**

Explanation: The query retrieves the minimum salary of all managers. This value then is assigned to all salesmen working in department 20.

Joining Relations

Comparisons in the where clause are used to combine rows from the tables listed in the from clause.

Example: In the table EMPLOYEE only the numbers of the departments are stored, not their name. For each salesman, we now want to retrieve the name as well as the number and the name of the department where he is working:

SELECT ENAME, E.DEPTNO, DNAME
FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO
AND JOB = 'SALESMAN';

Explanation: E and D are table aliases for EMP and DEPT, respectively. The computation of the query result occurs in the following manner (without optimization):

1. Each row from the table EMP is combined with each row from the table DEPT (this operationis called Cartesian product). If EMP contains m rows and DEPT contains n rows, we thus get **n x m** rows.

2. From these rows those that have the same department number are selected (where E.DEPTNO = D.DEPTNO).

3. From this result finally all rows are selected for which the condition JOB = 'SALESMAN' holds.

In this example the joining condition for the two tables is based on the equality operator "=".

The columns compared by this operator are called join columns and the join operation is called an **equijoin**.

Any number of tables can be combined in a select statement.

Example: For each project, retrieve its name, the name of its manager, and the name of the department where the manager is working:

```
select ENAME, DNAME, PNAME
from EMP E, DEPT D, PROJECT P
where E.EMPNO = P.MGR
and D.DEPTNO = E.DEPTNO;
```

It is even possible to join a table with itself:
Example: List the names of all employees together with the name of their manager:

```
select E1.ENAME, E2.ENAME
from EMP E1, EMP E2
where E1.MGR = E2.EMPNO;
```

Explanation: The join columns are MGR for the table E1 and EMPNO for the table E2.
The equijoin comparison is E1.MGR = E2.EMPNO.

The Comparison Operators

IN:which compares a value *v* with a set (or multiset) of values V and evaluates to TRUE if *v* is one of the elements in V.

The = **ANY** (or = **SOME**) operator returns **TRUE** if the value *v* is equal to some *value* in the set V and is hence equivalent to IN.

The keywords **ANY** and **SOME** have the same meaning. Other operators that can be combined with ANY (or SOME) include >,>=, <, <=, and < >. The keyword ALL can also be combined with each of these operators.

For example, the comparison condition *(v > ALL V)* returns TRUE if the value *v* is greater than *all* the values in the set (or multiset) V.

Retrieve all employees who are working in department 10 and who earn at least as much as any (i.e., at least one) employee working in department 30:

SELECT *
FROM emp
WHERE sal >= **ANY**
         (**SELECT** sal
         FROM emp
         WHERE deptno = 30)
   AND deptno = 10;

List all employees who are not working in department 30 and who earn more than all employees working in department 30:

Select * from EMP
where SAL > all
(select SAL from EMP
where DEPTNO = 30)
and DEPTNO <> 30;

For all and any, the following equivalences hold:

in : = any

not in : <> all or != all

Often a query result depends on whether certain rows do (not) exist in (other) tables. Such type of queries is formulated using the exists operator.

Example: List all departments that have no employees:

select * from DEPT

where not exists

(select *from EMP

where DEPTNO = DEPT.DEPTNO);

Basic Retrieval Queries in SQL

 SQL has one basic statement for retrieving information from a database.

The SELECT-FROM-WHERE clause Structure of Basic SQL Queries

The basic form of the SELECT statement, sometimes called a **mapping** or a **select-from-where block**, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

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

where **<attribute list>** is a list of attribute names whose values are to be retrieved by the query.
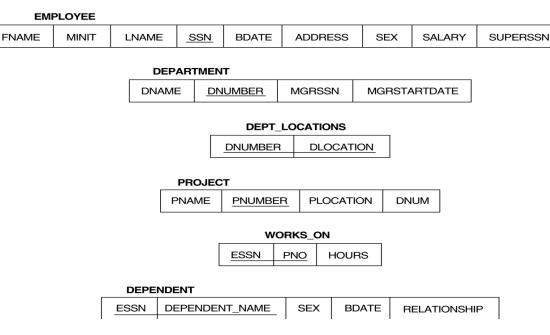
<table list> is a list of the relation names required to process the query.

<condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>.

 These correspond to the relational algebra operators =, <, ≤, >, ≥, and ≠, respectively, The main syntactic difference is the **not equal** operator.

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

CONSIDER THE FOLLOWING DETAILS:

EMPLOYEE(Fname,Minit,Lname,SSN,Bdate,Address,Sex,Salary,SuperSSN,Dno)
DEPARTMENT(Dname,Dnumber,MgrSSN,MgrStartDate)
DEPT_LOCATIONS(Dnumber,Dlocation)
PROJECT(Pname,pnumber,plocation,dnum)
WORKS_ON(essn,pno,hrs)
DEPENDENT(essn,dependentname,sex,bdate,relationship)

SQL Queries:

Query : Retrieve the birthdate and address of the employee whose name is 'John S Peter'.

SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME='John'
AND MINIT='S'
AND LNAME='Peter';

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

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

Query : For every project located in 'bangalore', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

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

Ambiguous Attribute Names, Aliasing,Renaming, and Tuple Variables

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in *different relations.* If this is the case, and a multitable query refers to two or more attributes with the same name, we *must* **qualify** the attribute name with the relation name to prevent ambiguity.

This is done by *prefixing* the **relation name to the attribute name** and separating the two by a **period**.
The Dno and Lname attributes of the EMPLOYEE relation were called Dnumber and Name, and the Dname attribute of DEPARTMENT was also called Name. then, to prevent ambiguity.
We must prefix the attributes Name and Dnumber in Q1A to specify which ones we are referring to, because the same attribute names are used in both relations:
Query 1A: Retrieve the name and address of all employees who work for the 'Research' department.
SELECT Fname, EMPLOYEE.Name, Address
FROM EMPLOYEE, DEPARTMENT
WHERE DEPARTMENT.Name='Research'
AND DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. We can also create an *alias* for each table name to avoid repeated typing of long table names .

Query: Retrieve the name and address of all employees who work for the 'Research' department.
SELECT EMPLOYEE.Fname, EMPLOYEE.LName, EMPLOYEE.Address
FROM EMPLOYEE, DEPARTMENT
WHERE DEPARTMENT.DName='Research'
AND DEPARTMENT.Dnumber=EMPLOYEE.Dno;

Query :For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.
SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE **AS** E,
EMPLOYEE **AS** S **WHERE** E.Super_ssn=S.Ssn;

In this case, we are required to declare alternative relation names E and S, called **aliases** or **tuple variables**.

Q:Retrieve the name and address of all employees who work for the 'Research' department.
SELECT Fname, Lname, Address
FROM EMPLOYEE, DEPARTMENT
WHERE Dname='Research' **AND** Dnumber=Dno;

SELECT E.Fname, E.LName, E.Address
FROM EMPLOYEE E, DEPARTMENT D
WHERE D.DName='Research' **AND** D.Dnumber=E.Dno;   (preffered)

Unspecified WHERE Clause and Use of the Asterisk

A *missing* WHERE clause indicates no condition on tuple selection. hence, *all tuples* of the relation specified in the FROM clause qualify and are selected for the query result.

If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—*all possible tuple combinations*—of these relations is selected.

Select all EMPLOYEE Ssns

SELECT Ssn
FROM EMPLOYEE;

Select all EMPLOYEE Ssns and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname in the database.

SELECT Ssn, Dname **FROM** EMPLOYEE, DEPARTMENT;

It is extremely important to specify every selection and join condition in the WHERE clause; if any such condition is overlooked, incorrect and very large relations may result.

OTHER EXAMPLES:

- **SELECT** * **FROM** EMPLOYEE **WHERE** Dno=5;

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

- **SELECT** * **FROM** EMPLOYEE, DEPARTMENT;

the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result.

In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not.

Q:Retrieve the salary of every employee

SELECT ALL Salary
FROM EMPLOYEE;

Q:Retrieve the salary of every employee and all distinct salary values

SELECT DISTINCT Salary

FROM EMPLOYEE;

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra.

There are set union **(UNION),** set difference (**EXCEPT**) and set intersection (**INTERSECT**) operations.

The relations resulting from these set operations are sets of tuples, that is, *duplicate tuples are eliminated from the result.* These set operations apply only to *union-compatible relations*, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations.

SQL supports three set operators which have the pattern:
<query 1> *<set operator>* **<query 2>**
The set operators are:
• **UNION [ALL:]** returns a table consisting of all rows either appearing in the result of   **<query 1>** or in the result of **<query 2>.**
Duplicates are automatically eliminated unless the clause all is used.

• **INTERSECT** returns all rows that appear in both results <query 1> and <query 2>.

• **MINUS** returns those rows that appear in the result of <query 1> but not in the result of <query 2>.

(a) Two tables, R(A) and S(A).
(b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

SQL also has corresponding multiset operations, which are followed by the keyword ALL (UNION ALL, EXCEPT ALL, INTERSECT ALL). Their results are multisets (duplicates are not eliminated). The behavior of these operations is illustrated by the examples in Figure .

The below example illustrates the use of UNION.

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

(**SELECT DISTINCT** Pnumber **FROM** PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=Dnumber **AND** Mgr_ssn=Ssn **AND** Lname='Smith')
                         UNION

(**SELECT DISTINCT** Pnumber **FROM** PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber=Pno **AND** Essn=Ssn **AND** Lname='Smith');

The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project.

Substring Pattern Matching and Arithmetic Operators In this section we discuss several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the **LIKE** comparison operator. This can be used for string **pattern matching**. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character.

Q. Retrieve all employees whose address is Houston, Texas.

SELECT Fname, Lname
FROM EMPLOYEE
WHERE Address **LIKE** '%Houston%';

Q . Find all employees who were born during the 1950s.

SELECT Fname, Lname
FROM EMPLOYEE
WHERE Bdate **LIKE** '_ _ 5 _ _ _ _ _ _ _';

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains.

 Q . Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

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

Another comparison operator, which can be used for convenience, is **BETWEEN Query .**

Q.Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

SELECT *
FROM EMPLOYEE
WHERE (Salary **BETWEEN** 30000 **AND** 40000) **AND** Dno = 5;

*The condition (Salary **BETWEEN** 30000 **AND** 40000) in above is equivalent to the condition ((Salary >= 30000) **AND** (Salary <= 40000)).*

Ordering of Query Results SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the **ORDER BY** clause.

Q . Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

SELECT D.Dname, E.Lname, E.Fname, P.Pname
FROM DEPARTMENT D, EMPLOYEE E, WORKS_ON W,PROJECT P
WHERE D.Dnumber= E.Dno
AND E.Ssn= W.Essn
AND W.Pno= P.Pnumber
ORDER BY D.Dname, E.Lname, E.Fname;

The keyword **ASC** can be used to specify ascending order explicitly.
For example, if we want descending alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause of can be written as **ORDER BY** D.Dname **DESC**, E.Lname **ASC**, E.Fname **ASC**

 Summary of Basic SQL Retrieval Queries A *simple* retrieval query in SQL can consist of up to four clauses, but only the first two—**SELECT** and **FROM**—are mandatory. The clauses are specified in the following order, with the clauses between square brackets […] being optional:

SELECT <attribute list>
FROM <table list>
[**WHERE** <condition>]
[**ORDER BY** <attribute list>];

 The SELECT clause lists the attributes to be retrieved, and the FROM clause specifies all relations (tables) needed in the simple query. The WHERE clause identifies the conditions for selecting the tuples from these relations, including join conditions if needed. ORDER BY specifies an order for displaying the results of a query.

Comparisons Involving NULL and Three-Valued Logic SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value *unknown* (exists but is not known), value *not available* (exists but is purposely withheld), or value *not applicable* (the attribute is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL.

1. Unknown value. A person's date of birth is not known, so it is represented by NULL in the database.

2. Unavailable or withheld value. A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database. **3. Not applicable attribute.** An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

It is often not possible to determine which of the meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish between the different meanings of NULL.

In general, each individual NULL value is considered to be different from every other NULL value in the various database records. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used.

Logical Connectives in Three-Valued Logic

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

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

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **IS** or **IS NOT**.

Q,Retrieve the names of all employees who do not have supervisors.

SELECT Fname, Lname
FROM EMPLOYEE
WHERE Super_ssn **IS** NULL;

Nested Queries, Tuples, and Set/Multiset Comparisons:
A nested query is a query that has another query embedded within it. The embedded query is called a subquery.
The embedded query can of course be a nested query itself; thus queries that have very deeply nested structures are possible. Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query**.

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

If a nested query returns a single attribute *and* a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

SELECT DISTINCT Essn
FROM WORKS_ON

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

This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn ='123456789') works on. In this example, the IN operator compares the subtuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > **ALL** (**SELECT** Salary
                FROM EMPLOYEE
                WHERE Dno=5);

Correlated Nested Queries In the nested queries seen thus far, *the inner subquery has been completely independent of the outer query*. In general, the inner subquery could depend on the row currently being examined in the outer query

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

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

Q: Retrieve the name of each employee who has a dependent with the same first name as the employee.

SELECT E.FNAME, E.LNAME
FROM EMPLOYEE **AS** E
WHERE E.SSN **IN (SELECT** ESSN
                FROM DEPENDENT

                                    WHERE ESSN=E.SSN **AND**
E.FNAME=DEPENDENT_NAME**)**

or

Q: SELECT E.FNAME, E.LNAME
FROM EMPLOYEE E, DEPENDENT D
WHERE E.SSN=D.ESSN **AND** E.FNAME=D.DEPENDENT_NAME **;**

The **CONTAINS** operator compares two *sets of values* , and returns TRUE if one set contains all values in the other set (reminiscent of the *division* operation of algebra).

Q: Retrieve the name of each employee who works on *all* the projects controlled by department number 5.

SELECT FNAME,LNAME
FROM EMPLOYEE
WHERE ((SELECT PNO
         FROM WORKS_ON
WHERE SSN=ESSN)
CONTAINS
            (SELECT PNUMBER
             FROM PROJECT
             WHERE DNUM=5))
In the above method, the second nested query which is not correlated with the outer query, retrieves the project numbers of all projects controlled by department 5.
The first nested query, which is correlated, retrieves the project numbers on which the employee works, which is different *for each employee tuple* because of the correlation.

The EXISTS and NOT EXISTS Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated nested query is *empty* (contains no tuples) or not.

The result of EXISTS is a Boolean value **TRUE** if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples.

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

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

Or

SELECT E.Fname, E.Lname
FROM EMPLOYEE E, DEPENDENT D
              WHERE E.Ssn=D.Essn **AND** E.Sex=D.Sex **AND**
                    E.Fname=D.Dependent_name; **Or**
Q: SELECT E.Fname, E.Lname
FROM EMPLOYEE **AS** E
WHERE EXISTS
(**SELECT** *
FROM DEPENDENT **AS** D
WHERE   E.Ssn=D.Essn   **AND**   E.Sex=D.Sex   **AND**
E.Fname=D.Dependent_name);
EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query.
Q. Retrieve the names of employees who have no dependents.
 SELECT Fname, Lname
FROM EMPLOYEE
WHERE NOT EXISTS (**SELECT** *
                     FROM DEPENDENT
                       WHERE Ssn=Essn);
In above correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If *none exist,* the EMPLOYEE tuple is selected because the **WHERE**-clause condition will evaluate to **TRUE** in this case. We can explain Query as follows:

For each EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn, if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

Q. List the names of managers who have at least one dependent.

SELECT Fname, Lname
FROM EMPLOYEE
WHERE EXISTS (**SELECT** *
         FROM DEPENDENT
         WHERE Ssn=Essn)**AND**
                 EXISTS (**SELECT** *
                       FROM DEPARTMENT
                       WHERE Ssn=Mgr_ssn);

It is possible to write the above query by using single nested query or without using nested query.

Q :Retrieve the name of each employee who works on all *the projects controlled by department number 5 (it* can be written using EXISTS and NOT EXISTS in SQL systems.)

SELECT Fname, Lname
FROM EMPLOYEE
WHERE NOT EXISTS ((**SELECT** Pnumber
                 FROM PROJECT
                 WHERE Dnum=5) **EXCEPT**
                           (**SELECT** Pno
                         FROM WORKS_ON
                        WHERE Ssn=Essn));

or

SELECT FNAME, LNAME
FROM EMPLOYEE
WHERE ((SELECT PNO
        FROM WORKS_ON
        WHERE SSN=ESSN**)**
 CONTAINS
        (SELECT PNUMBER
         FROM PROJECT
         WHERE DNUM=5**));**

Or

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

Aggregate Functions in SQL:

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary.

Grouping is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database applications.

A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.

The COUNT function returns the number of tuples or values as specified in a query. The functions **SUM, MAX, MIN, and AVG** can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values.

These functions can be used in the SELECT clause or in a HAVING clause (which we introduce end of the topic).

The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.

Q. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

SELECT SUM (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)
FROM EMPLOYEE;

Q. Find the sum of the salaries of all Research department employees, the maximum salary, the minimum salary, and the average salary.

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

Q. Retrieve the total number of employees in the company

SELECT COUNT (*) **FROM** EMPLOYEE;

Q; Retrieve the total number of employees in the company and the number of employees in the 'Research' department .

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

Here the *asterisk (*) refers to the rows (tuples*), so **COUNT (*)** returns the number of rows in the result of the query. We may also use the *COUNT function to count values in a column rather than tuples, as in the next example.*

Q . Count the number of distinct salary values in the database.

SELECT COUNT (**DISTINCT** Salary)
FROM EMPLOYEE;

Q: Retrieve the names of all employees who have two or more dependents

Q: SELECT Lname, Fname
FROM EMPLOYEE
WHERE (**SELECT COUNT** (*)
        FROM DEPENDENT
        WHERE Ssn=Essn) >= 2;

The correlated nested query counts the number of dependents that each employee has. If this is greater than or equal to two, the employee tuple is selected.

Grouping: The GROUP BY and HAVING Clauses

The GROUP BY clause specifies the grouping attributes, which should *also appear in the SELECT clause,* so that the value resulting from applying each

aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

Q. For each department, retrieve the department number, the number of employees in the department, and their average salary.
SELECT Dno, **COUNT** (*), **AVG** (Salary)
FROM EMPLOYEE **GROUP BY** Dno;

Figure
Results of GROUP BY

(a)

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

Grouping EMPLOYEE tuples by the value of Dno

Q. For each project, retrieve the project number, the project name, and the number of employees who work on that project.

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

The above Query shows how we can use a join condition in conjunction with GROUP BY.
In this case, the grouping and functions are applied *after* the joining of the two relations. SQL provides a **HAVING** clause, which can appear in conjunction with a **GROUP BY** clause, for this purpose.
HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query.

Q. For each project *on which more than two employees work,* retrieve the project number, the project name, and the number of employees who work on the project.

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

the HAVING clause serves to choose *whole groups.*

Q . For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.
SELECT Pnumber, Pname, **COUNT** (*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber=Pno **AND** Ssn=Essn **AND** Dno=5
GROUP BY Pnumber, Pname;

Q. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.
SELECT Dnumber, **COUNT** (*)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber=Dno **AND** Salary>40000 **AND**
             (**SELECT** Dno
              FROM EMPLOYEE
              GROUP BY Dno
              HAVING COUNT (*) > 5)

A retrieval query in **SQL can consist of up to six clauses**, but only the first two— **SELECT and FROM—are mandatory**.
The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way.
The clauses are specified in the following order, with the clauses between square brackets […] being optional:

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

The SELECT clause lists the attributes or functions to be retrieved.
The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries.
The WHERE clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed.

GROUP BY specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. *Finally, ORDER BY specifies an order for displaying the result of a query.*

In general, there are numerous ways to specify the same query in SQL. This flexibility in specifying queries has advantages and disadvantages. The main advantage is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the IN comparison operator. Some users may be more comfortable with one approach, whereas others may be more comfortable with another. From the programmer's and the system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible. The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way. Ideally, this should not be the case: The DBMS should process the same query in the same way regardless of how the query is specified. But this is quite difficult in practice, since each DBMS has different methods for processing queries specified in different ways.

Sailors(<u>sid</u>: integer, sname: string, rating: integer, age: real);
Boats(<u>bid</u>: integer, bname: string, color: string);
Reserves(<u>sid:</u> integer<u>, bid</u>: integer, day: date).

Find all information of sailors who have reserved boat number 103.
SELECT S.*
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103;
Or
SELECT **Sailors.***
FROM Sailors, Reserves
WHERE Sailors.sid = Reserves.sid AND Reserves.bid = 103;
* can be used if you want to retrieve all columns.

Find the names of sailors who have reserved a red boat, and list in the order of age.
SELECT S.sname, S.age

FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
ORDER BY S.age
ORDER BY S.age [ASC] (default)
ORDER BY S.age DESC

 Find the names of sailors who have reserved at least one boat.
SELECT sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
The join of Sailors and Reserves ensure that for each select sname, the sailor has made some reservation.
Find the ids and names of sailors who have reserved two different boats on the same day.
SELECT DISTINCT S.sid, S.sname
FROM Sailors S, Reserves R1, Reserves R2
WHERE S.sid = R1.sid AND S.sid = R2.sid
AND R1.day = R2.day AND R1.bid <> R2.bid;
Using Expressions and Strings in the SELECT Command.
SELECT sname, age, rating + 1 as sth
FROM Sailors
WHERE 2* rating – 1 < 10 AND sname like **'B_%b';**
SQL provides for pattern matching through LIKE operator, along with the use of symbols:
% (which stands for **zero or more** arbitrary characters) and
_ (which stands for **exactly one**, arbitrary, characters)
Union, Intersect and Except
Note that Union, Intersect and Except can be used on only two tables that are **union-compatible**,
that is, have the same number of columns and the columns, taken in order, have the same types.
Ex. Find the ids of sailors who have reserved a red boat or a green boat.
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
UNION
SELECT R2.sid
FROM Boats B2, Reserves R2
WHERE R2.bid = B2.bid AND B2.color = 'green'

The default for UNION queries is that **duplicates *are* eliminated**. To retain duplicates, use UNION ALL.
Replace UNION with **UNION ALL**.
Replace UNION with **INTERSECT**.
Replace UNION with **EXCEPT**.

Nested Query
IN and **NOT IN**
EXISTS and **NOT EXISTS**
UNIQUE and **NOT UNIQUE**
op ANY
op ALL
Find the names of sailors who have reserved boat 103.
SELECT S.sname
FROM Sailors S
WHERE S.sid **IN** (SELECT R.sid
FROM Reserves R
WHERE R.bid = 103)
The inner subquery has been completely independent of the outer query.
(Correlated Nested Queries)
SELECT S.sname
FROM Sailors S
WHERE **EXISTS** (SELECT *
FROM Reserves R
WHERE R.bid = 103
AND **R.sid = S.sid**)
The inner query depends on the row that is currently being examined in the outer query.
Find the name and the age of the youngest sailor.
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age **<= ALL** (SELECT age
FROM Sailors)
Find the names and ratings of sailor whose rating is better than some sailor called Horatio.
SELECT S.sname, S.rating
FROM Sailors S
WHERE S.rating **> ANY** (SELECT S2.rating
FROM Sailors S2
WHERE S2.sname = 'Horatio')

Note that **IN** and **NOT IN** are equivalent to **= ANY** and **<> ALL**, respectively.
Find the names of sailors who have reserved all boats.
SELECT S.sname
FROM Sailors S
WHERE **NOT EXISTS** ((SELECT B.bid
FROM Boats B)
EXCEPT
(SELECT R.bid
FROM Reserves R
WHERE **R.sid = S.sid**))

An alternative solution:
SELECT S.sname
FROM Sailors S
WHERE **NOT EXISTS** (SELECT B.bid
FROM Boats B
WHERE **NOT EXISTS** (SELECT R.bid
FROM Reserves R
WHERE R.bid = B.bid
AND **R.sid = S.sid**))
Aggregation Operators
Count the number of different sailor names.
SELECT COUNT(DISTINCT S.sname)
FROM Sailors S
Calculate the average age of all sailors.
SELECT AVG(s.age)
FROM Sailors S
Find the name and the age of the youngest sailor.
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = (SELECT MIN(S2.age)
FROM Sailors S2)
SELECT [DISTINCT] select-list
FROM from-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification

Find the average age of sailors for each rating level.
SELECT S.rating, AVG(S.age) AS avg_age
FROM Sailors S
GROUP BY S.rating;

Find the average age of sailors for each rating level that has at least two sailors.
SELECT S.rating, AVG(S.age) AS avg_age
FROM Sailors S
GROUP BY S.rating
HAVING COUNT(*) > 1;

Sailors

Sid	Sname	Rating	Age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	40
85	Art	3	25.5
95	Bob	3	63.5

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Reserves

sid	bid	day
22	101	1998-10-10
22	102	1998-10-10
22	103	1998-10-8
22	104	1998-10-7
31	102	1998-11-10
31	103	1998-11-6
31	104	1998-11-12
64	101	1998-9-5
64	102	1998-9-8
74	103	1998-9-8

An example shows difference between WHERE and HAVING:

SELECT S.rating, AVG(S.age) as avg_age
FROM Sailors S
WHERE S.age >=40
GROUP BY S.rating

Rating	avg_age
3	63.5
7	45
8	55.5

SELECT S.rating, AVG(S.age) as avg_age
FROM Sailors S
GROUP BY S.rating
HAVING AVG(S.age) >= 40

Rating	avg_age
3	44.5
7	40
8	40.5