Contents

Basic SQL Query

```
SELECT [DISTINCT] `target-list`
FROM `relation-list`
WHERE `qualification`
```

- relation-list: A list of relation names(possibly with a range-variable after each name).
- target-list: A list of attributes of relations in relation-list.
- qualification: Comparisons combined using AND, OR and NOT.
- DISTINCT: An optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!

example1

case: Find the names of students who applied to Stanford.

```
SELECT S.name
FROM `Student` `S`, `Apply` `A`
WHERE S.sID = A.sID AND A.cName = 'Stanford'
```

- `S` and `A` are range variables. `S` is used to refer to `Student` relation and `A` is used to refer to `Apply` relation.

example2

case: Find triples (of ages of students and two filelds defined by arithmetic expressions) for students whose names begin with B and end with B and contain at least three characters.

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM `Student` `S`
WHERE S.sname LIKE `B_%B`
```

- Illustrates use of arithmetic expressions and string pattern matching: Find triples (of ages of students and two filelds defined by arithmetic expressions) for students whose names begin with B and end with B and contain at least three characters.
- `AS` and `=` are two ways to name fields in result.
- `LIKE` is used for string matching. `_` matches any single character, `%` matches 0 or more arbitrary characters.

example3

case: Find IDs of sailors who have reserved a red or green boat.

```
SELECT S.sid
FROM `Sailors` `S`, `Boats` `B`, `Reserves` `R`
WHERE S.sid = R.sid AND R.bid = B.bid AND (B.color = 'red' OR B.color = 'green')
```

or you can also use `UNION`:

```
SELECT S.sid
FROM `Sailors` `S`, `Boats` `B`, `Reserves` `R`
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT S.sid
FROM `Sailors` `S`, `Boats` `B`, `Reserves` `R`
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'green'
```

- `UNION`: Can be used to compute the union of any two **union-compatible** sets of tuples (which are themselves the result of SQL queries).
- `OR`, `AND`, `NOT`: Can be used to combine comparisons.

example4

case: Find IDs of sailors who have reserved a red boat but not a green boat.

```
SELECT S.sid
FROM `Sailors` `S`, `Boats` `B`, `Reserves` `R`
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT S.sid
FROM `Sailors` `S`, `Boats` `B`, `Reserves` `R`
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'green'
```

- `EXCEPT`: Can be used to compute the set difference of any two **union-compatible** sets of tuples.

example5

case: Find IDs of sailors who have reserved a red boat and a green boat.

```
SELECT S.sid
FROM Sailors S,
     Boats B1, Reserves R1,
     Boats B2, Reserves R2
WHERE S.sid = R1.sid AND R1.bid = B1.bid
  AND S.sid = R2.sid AND R2.bid = B2.bid
  AND (B1.color = 'red' AND B2.color = 'green')
```

or you can also use `INTERSECT`:

```
SELECT S.sid
FROM Sailors S,
     Boats B1, Reserves R1
WHERE S.sid = R1.sid AND R1.bid = B1.bid AND B1.color = 'red'
INTERSECT
SELECT S.sid
FROM Sailors S,
     Boats B2, Reserves R2
WHERE S.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

- `INTERSECT`: Can be used to compute the intersection of any two **union-compatible** sets of tuples.

example6

case: Find names of sailors who have reserved a red boat. (**Nested Queries**)

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R, Boats B
                WHERE R.bid = B.bid AND B.color = 'red')
```

- Nested Queries: The inner query is evaluated first and its result is used in the outer query.
- `IN`: Can be used to test if a value is in a set of values.
- `NOT IN`: Can be used to test if a value is not in a set of values.

case: Find names of sailors who have reserved a red boat. (**Nested Queries with Correlation**)

```
SELECT S.sname
FROM   Sailors S
WHERE EXISTS (SELECT *
              FROM Reserves R
              WHERE R.bid = 103 AND S.sid = R.sid)
```

- Nested Queries with Correlation
- `ESISTS`: is another set comparison operator, like `IN`. Also, `NOT EXISTS` can be used.

example7

case: Find IDs of boats which are reserved by only one sailor.

```
SELECT bid
FROM Reserves R1
WHERE bid NOT IN (SELECT bid
                  FROM Reserves R2
                  WHERE R1.sid <> R2.sid)
```

example8

case: Find sailors whose rating is greater than that of some sailror called Horatio.

```
SELECT *
FROM Sailors S
WHERE S.rating > (SELECT S2.rating
                  FROM Sailors S2
                  WHERE S2.sname = 'Horatio')
```

- `ANY`: Can be used to compare a value with a set of values.
- `ALL`: Can be used to compare a value with a set of values.

example9

case: Find sid' s of sailors who' ve reserved both a red and a green boat.

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
      AND S.sid IN (SELECT S2.sid
                    FROM Sailors S2, Boats B2, Reserves R2
                    WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green')
```

example10

case: Find sailors who' ve 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))
```

- `EXCEPT`: Can be used to compute the set difference of any two **union-compatible** sets of tuples.

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid  -- Sailors S such that there is no
                  FROM Boats B
                  WHERE NOT EXISTS (SELECT R.bid  -- ... boat B without ...
                                    FROM Reserves R
                                    WHERE R.sid = S.sid AND R.bid = B.bid))
```

Aggregate Operators

- Significant extension of relational algebra.
  - `COUNT (*)`
  - `COUNT ([DISTINCT] A)`
  - `SUM ([DISTINCT] A)`
  - `AVG ([DISTINCT] A)`
  - `MAX (A)`
  - `MIN (A)`
- A is single column (an attribute)
- ATTENTION: Aggregate operators can not be used nested in other aggregate operators.

example1

```
SELECT COUNT (*)
FROM Sailors S
```

```
SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname = 'Bob'
```

```
SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT AVG (DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT S.sname
FROM Sailors S
WHERE S.rating = (SELECT MAX (S2.rating)
                  FROM Sailors S2)
```

example2

case: Find name and age of the oldest sailor.

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = (SELECT MAX (S2.age)
               FROM Sailors S2)
```

example3

```
SELECT [DISTINCT] `target-list`
FROM              `relation-list`
WHERE             `qualification`
GROUP BY          `grouping-list`
HAVING            `group-qualification`
```

- The target-list contains
  - (i) attribute names
  - (ii) terms with agregate operations
- The attribute list (i) must be a subset of `grouping-list`. Intuitively, each answer tuple corresponds to a group, and these attrributes must have a single value per group. (A `group` is a set of tuples that have the same value for all attributes in `grouping-list`.)
- The cross-product of `relation-list` is computed, tuples that fail `qualification` are discarded, 'unnecessary' fields are deleted, and the reamining tuples are partitioned into groups by the value of attributes in `grouping-list`. The `group-qualification` is then applied to eliminate some groups. Expressions in group-qualification must have a **single value per group**. One answer tuple is generated per qualifying group.

case: Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) >= 2
```

case: Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors and with every sailor under 60

```
...
HAVING COUNT(*) > 1 AND EVERY (S.age < 60)
```

example4

case: For each red boat, find the number of reservations for this boat

```
SELECT B.bid, COUNT (*) AS scount
FROM Boats B, Reserves R
WHERE B.bid = R.bid AND B.color = 'red'
GROUP BY B.bid
```

you can also use `HAVING`:

```
SELECT B.bid, COUNT (*) AS scount
FROM Boats B, Reserves R
WHERE B.bid = R.bid
GROUP BY B.bid, B.color = 'red'
HAVING B.color = 'red'
```

example5

case: Find age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age > 18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT (*)
            FROM Sailors S2
            WHERE S2.rating = S.rating)
```

- shows that `HAVING` clause can contain a sub-query.

example6

case: Find those ratings for which the average age is the minimum over all ratings.

```
SELECT Temp.rating
FROM (SELECT S.rating, AVG(S.age) AS avgage
      FROM Sailors S
      GROUP BY S.rating) AS Temp
WHERE Temp.avgage = (SELECT MIN (Temp.avgage)
                     FROM Temp)
```

NULL Values

- Field values in a tuple are sometimes unknown or inapplicable.
  - SQL provides a special value `NULL` for such cases.

Some New Features of SQL

CAST expression

- change the expression to the target data type
- use:
  - match function parameters
    - substr(string1, CAST(x AS Integer), CAST(y AS Integer))
  - change precision while calculating

        * CAST(elevation AS DECIMAL(5,0))
    - assign a data type to NULL value

example

Students(name, school) Soldiers(name, service)

```
CREATE VIEW `prospects` (`name`, `school`, `service`) AS
    SELECT `name`, `school`, CAST(NULL AS VARCHAR(20))
    FROM `Students`
UNION
    SELECT `name`, CAST(NULL AS VARCHAR(20)), `service`
    FROM `Soldiers`
```

CASE expression

example

Officers(name, status, rank, title)

```
SELECT `name`, CASE `status`
                WHEN 1 THEN 'active'
                WHEN 2 THEN 'retired'
                ELSE 'unknown'
                END AS `status`
FROM `Officers`l
```

example

Machines(serialno, type, year, hours_used, accidents)

case: Find the rate of accidents of "chain saw" in the whole accidents:

```
SELECT sum(CASE `type`
           WHEN 'chain saw' THEN `accidents`
           ELSE 0e0
           END) / sum(`accidents`)
FROM `Machines`;
```

example

case: Find the average accident rate of every kind of equipment:

```
SELECT `type`, CASE
                   WHEN sum(`hours_used`) = 0 THEN 0e0
                   ELSE sum(`accidents`) / sum(`hours_used`)
               END AS `accident_rate`
FROM `Machines`
GROUP BY `type`;
```

or you can also:

```
SELECT `type`, sum(`accidents`) / sum(`hours_used`) AS `accident_rate`
FROM `Machines`
GROUP BY `type`
HAVING sum(`hours_used`) > 0;
```

Sub-query

- The functions of sub-queries have been enhanced in new SQL standard. Now they can be used in SELECT and FROM clauses.
    - Scalar sub-query
    - Table expression
    - Common table expression

Scalar Sub-query

- The result of a sub-query is a single value. It can be used in the place where a value can orrur.

example

case: Find the departments whose average bonus is higher than average salary:

```
SELECT d.deptname, d.location
FROM dept AS d
WHERE (SELECT AVG(bonus)
      FROM emp
      WHERE deptno = d.deptno) >
      (SELECT AVG(salary)
      FROM emp
      WHERE deptno = d.deptno);
```

case: List the deptno, deptname, and the max salary of all departments located in New York:

```
SELECT d.deptno, d.deptname, (SELECT MAX(salary)
                             FROM emp
                             WHERE deptno = d.deptno) AS max_salary
FROM dept AS d
WHERE d.location = 'New York';
```

Table Expression

- The result of a sub-query is a table. It can be used in the place where a table can occur.

example

```
SELECT `startyear`, avg(pay)
FROM (SELECT `name`, `salay`+`bonus` AS pay, YEAR(startdate) AS `startyear`
      FROM `emp`) AS temp
GROUP BY `startyear`;
```

case: Find departments whose total payment is greater than 200000

```
SELECT `deptno`, `totalpay`
FROM (SELECT `deptno`, SUM(`salay`) + SUM(`bonus`) AS `totalpay`
      FROM `emp`
      GROUP BY `deptno`) AS `payroll`
WHERE `totalpay` > 200000;
```

Common Table Expression

- In some conplex query, a table expression may need occurring more than one time in the same SQL statements. Although it is permitted, the efficiency is low and there maybe inconsistency problem.
- `WITH` clause can be used to define a common table expression. In fact, it defines a temporary view.

example

case: Find the department who has the highest total payment:

```
WITH payroll(deptno, totalpay) AS
    (SELECT `deptno`, SUM(salary) + SUM(bonus)
    FROM `emp`
    GROUP BY `deptno`)
SELECT `deptno`
FROM `payroll`
WHERE `totalpay` = (SELECT MAX(`totalpay`)
                    FROM `payroll`);
```

case: Find department pairs, in which the first department' s average salary is more than two times of the seoncd one' s:

```
WITH deptavg(deptno, avgsal) AS
    (SELECT `deptno`, AVG(`salary`)
    FROM `emp`
    GROUP BY `deptno`)
SELECT d1.deptname, d1.avgsal, d2.deptname, d2.avgsal
FROM `deptavg` AS d1, `deptavg` AS d2
WHERE d1.avgsal > 2 * d2.avgsal AND d1.deptno <> d2.deptno;
```

Outer Join

example

Teacher(name, rank) Course(subject, enrollment, quarter, teacher)

```
WITH
  innerjoin(`name`, `rank`, `subject`, `enrollment`) AS
    (SELECT t.name, t.rank, c.subject, c.enrollment
    FROM `Teacher` AS t, `courses` AS c
    WHERE t.name = c.teacher AND c.quarter = 'Fall 96'),
  teacher-only(`name`, `rank`) AS
    (SELECT `name`, `rank`
    FROM `Teacher`
    EXCEPT ALL
    SELECT `name`,`rank`
    FROM `innerjoin`),
  course-only(`subject`, `enrollment`) AS
    (SELECT `subject`, `enrollment`
    FROM `courses`
    EXCEPT ALL
    SELECT `subject`, `enrollment`
    FROM `innerjoin`)
  SELECT `name`, `rank`, `subject`, `enrollment`
  FROM `innerjoin`
  UNION ALL
  SELECT `name`, `rank`,
      CAST(NULL AS VARCHAR(20)) AS `subject`,
      CAST(NULL AS INTEGER) AS `enrollment`
  FROM `teacher-only`
  UNION ALL
  SELECT CAST(NULL AS VARCHAR(20)) AS `name`,
      CAST(NULL AS VARCHAR(20)) AS `rank`,
      `subject`, `enrollment`
  FROM `course-only`;
```

- `EXCEPT ALL`: can be used to compute the set difference of any two **union-compatible** sets of tuples.
- `UNION ALL`: can be used to compute the union of any two **union-compatible** sets of tuples (which are themselves the result of SQL queries).

Recursion

- If a common table expression uses itself in its definition, this is called recursion. It can calculate a compolex recursive inference in one SQL statement.

example

FedEmp(name, salary, manager)

case: Find all employees under the management of Hoover and whose salary is more than 100000

```
WITH agents(`name`, `salary`) AS
    ((SELECT `name`, `salary`   -- initial query
      FROM `FedEmp`
      WHERE manager = 'Hoover')
    UNION ALL
     (SELECT f.name, f.salary   -- recursive query
      FROM `agents` AS a, `FedEmp` AS f
      WHERE f.manager = a.name))
SELECT `name`                   -- final query
FROM `agents`
WHERE `salary` > 100000;
```

case: Find how much rivets are used in one wing

```
WITH wingpart(subpart, qty) AS
        ((SELECT `subpart`, `qty`   -- initial query
          FROM `Parts`
          WHERE `part` = 'wing')
    UNION ALL
        (SELECT p.subpart, p.qty    -- recursive query
        FROM `wingpart` AS w, `Parts` AS p
        WHERE p.part = w.subpart))
SELECT SUM(`qty`) AS qty            -- final query
FROM `wingpart`
WHERE `subpart` = 'rivet';
```

case: Find all subparts and their total quantity needed to assemble a wing:

```
WITH wingpart(subpart, qty) AS
        ((SELECT `subpart`, `qty`   -- initial query
          FROM `Parts`
          WHERE `part` = 'wing')
    UNION ALL
        (SELECT p.subpart, p.qty    -- recursive query
        FROM `wingpart` AS w, `Parts` AS p
        WHERE p.part = w.subpart))
SELECT `subpart`, SUM(qty) AS qty    -- final query
FROM `wingpart`
GROUP BY `subpart`;
```

Data Manipulation Language

- Insert
  - Insert a tuple into a table
- Delete
  - Delete tuples fulfill qualifications
- Update
  - Update the attributes' value of tuples fulfill

View in SQL

- General view
  - Virtual tables derived from base tables
  - Logical data independence
  - Security of dat
  - Update problems of view
- Temporary view and recursive query

  - WITH
  - RECURSIVE

Update problems of view

```
CREATE VIEW YoungSailor AS
    SELECT sid, sname, rating
    FROM Sailors
    WHERE age < 30;

CREATE VIEW Ratingavg AS
    SELECT rating, AVG(age)
    FROM Sailors
    GROUP BY rating;
```

Embedded SQL

- In order to access database in programs, and take further process to the query results, need to combine SQL and programming language (such as C, C++, Java, etc.)
- Problems should be solved:
  - How to accept SQL statements in programming language
  - How to exhchange data and messages between programming language and DBMS
  - The query result of DBMS is a set, how to transfer it to the variables in programming language
  - The data type of DBMS and programming language may not the same exactly

General Solution

- Embedded SQL
- Programming APIs
- Class Library