四川大學
SICHUAN UNIVERSITY

Database System Concepts

Introduction to SQL

伍元凯
College of Computer Science (Software), Sichuan University
wuyk0@scu.edu.cn

2023/03/08

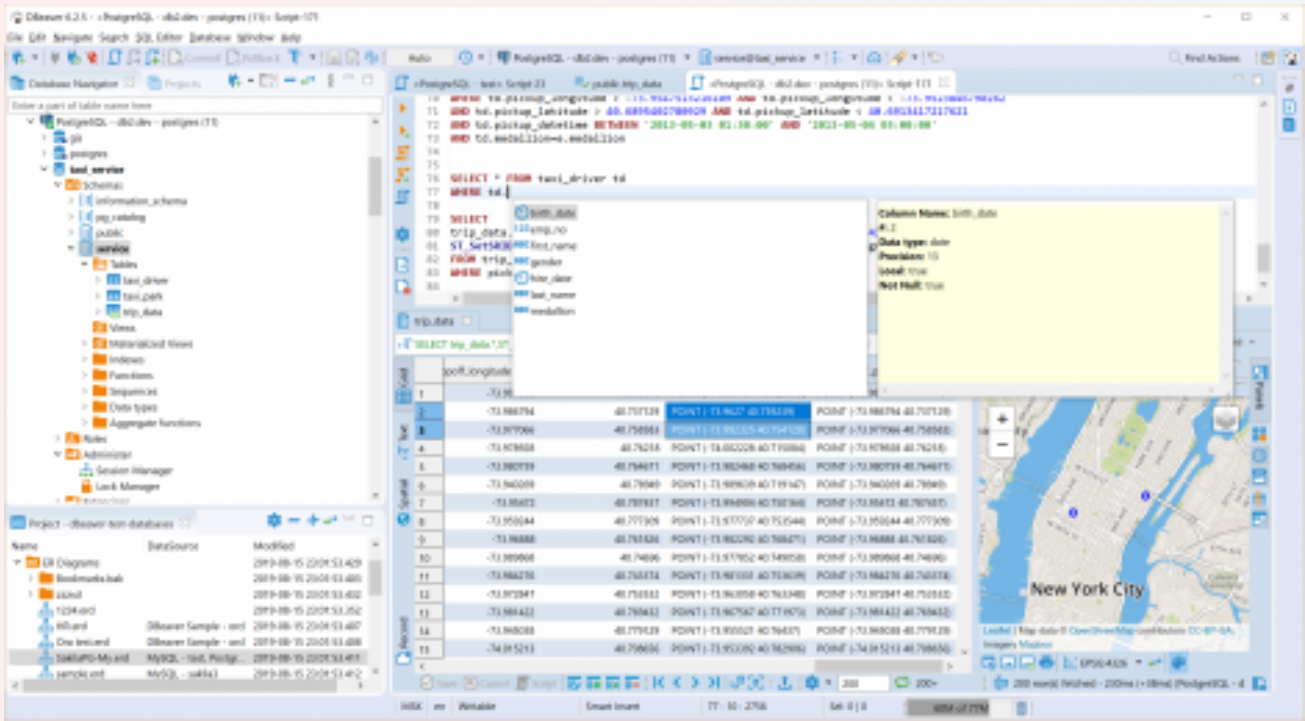
- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory,Renamed Structured Query Language (SQL，结构化查询语言)
- ANSI and ISO standard SQL: SQL-86, SQL-89, SQL-92 SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016.
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- Not all examples here may work on your particular system.

The course's intention is to present SQL's fundamental constructs and concepts. Your system may differ in details or may support only a subset. Some online platform for practicing: **SQLZoo**[1] and **HackerRank**[2]

[1] https://sqlzoo.net/wiki/SQL_Tutorial

[2] https://www.hackerrank.com/challenges/revising-the-select-query

1. Download MySQL Community Server from https://dev.mysql.com/downloads/mysql/.
2. Download DBeaver Community Edition from https://dbeaver.io/download/.



Just for recommendation, you can use any other types of SQL

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The domain (basic type) of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

char(n). Fixed length character string, with user-specified length n.

varchar(n). Variable length character strings, with user-specified maximum length n.

int. Integer (a finite subset of the integers that is machine-dependent).

smallint. Small integer (a machine-dependent subset of the integer domain type).

numeric(p,d). Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point. **numeric(3,1)** allows 44.5 to be store exactly, but neither 444.5 or 0.32

real, double precision. Floating point and double-precision floating point numbers, with machine-dependent precision.

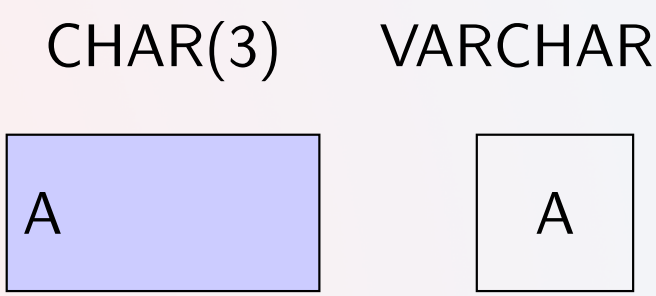
float(n). Floating point number, with user-specified precision of at least n digits.

Each type may include a special value called **null** value. **null** values can be used to represent situations where data is missing or unknown.

ID	Name	Age
1	John	25
2	Jane	NULL
3	Michael	30

Example table with NULL values

In certain cases, we may wish to prohibit **null** value.



- The **CHAR(n)** data type is fixed-length, meaning that it always allocates a specific number of bytes for each entry, regardless of the actual length of the string (social security numbers or phone numbers).
- The **VARCHAR(n)** data type is variable-length, which means that it only allocates the amount of storage needed to store the actual data (name or addresses).
- For 汉字, we can use **NVARCHAR**. NVARCHAR can store Unicode characters that supports a much wider range of characters, including characters used in many non-Latin languages.

目录

General From of create table

```
1 CREATE TABLE table_name (
2 column1 datatype1 [NULL | NOT NULL],
3 column2 datatype2 [NULL | NOT NULL],
4 column3 datatype3 [NULL | NOT NULL],
5 ...
6 PRIMARY KEY (one or more columns)
7 FOREIGN KEY (one or more columns) REFERENCES \
8 table_name(column_name)
9 );
```

Some integrity constraints in this section:

- Primary key: A primary key is a column or set of columns that uniquely identifies each row in a table. It ensures that each row in a table is **unique**.
- Foreign key: A foreign key is a column or set of columns that refers to a primary key in another table. It ensures that the data in a table is **consistent and accurate by enforcing referential integrity**.
- NOT NULL: This constraint helps to ensure data integrity by preventing the insertion of null values into columns where a value is required

Data with integrity constraints

```sql
CREATE TABLE customers (
  id INT PRIMARY KEY,
  name VARCHAR(50)
);
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT,
  order_date DATE,
  FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

a customer_id value that doesn't exist in the customers table

```sql
INSERT INTO orders (id, customer_id, order_date)
VALUES (1, 99, '2022-03-01');
```

error flags

```
ERROR:  insert or update on table "orders" violates foreign key
      constraint "orders_customer_id_fkey"
DETAIL:  Key (customer_id)=(99) is not present in table "customers"
      .
```

Relational Algebra	SQL
minimal superkey can uniquely identify each tuple in a relation.	**one or more attributes** can uniquely identify each tuple in a relation.
A relation can only have one primary key.	A relation can have multiple primary keys.
no additional attributes or properties.	additional attributes or properties, such as auto-increment and foreign key references.

Two methods for removing data:

Drop

```
1 DROP TABLE employees;
```

Delete

```
1 DELETE FROM employees WHERE age > 50;
```

- **DROP TABLE** is used to remove an entire table from the database.
- **DELETE FROM** is used to remove one or more rows from a table.

Add/remove attributes:

```
1 customers (id INT, name VARCHAR(50), email VARCHAR(50))
```

Add phone attribute

```
1 ALTER TABLE customers ADD COLUMN phone VARCHAR(20);
```

```
1 customers (id INT, name VARCHAR(50), email VARCHAR(50), phone
      VARCHAR(20))
```

Delete phone attribute

```
1 ALTER TABLE customers DROP COLUMN phone;
```

```
1 customers (id INT, name VARCHAR(50), email VARCHAR(50))
```

A real-world query example[3]

[3] https://help.aliyun.com/document_detail/161461.html

Table for practicing

id	name	age	city
1	John	25	New York
2	Sarah	30	Los Angeles
3	Alex	28	Chicago
4	Mary	35	Houston
5	Peter	27	San Francisco
6	John	25	New York
7	Alex	28	Chicago

Example table for practicing basic SQL queries: sql_practice

目录

```
1 SELECT city FROM sql_practice;
```

The corresponding relational algebra expression for this SELECT statement is:

π_{city}(sql_practice)

city
New York
Los Angeles
Chicago
Houston
San Francisco
New York
Chicago

Resulting table

We can use the SELECT DISTINCT statement to retrieve only unique values from a table.

```
1 SELECT DISTINCT city FROM sql_practice;
```

city
New York
Los Angeles
Chicago
Houston
San Francisco

Resulting table

Select can work with +, -, *, /:

```
1 SELECT id, name, age + 1 as age, city FROM sql_practice;
```

Resulting Table:

id	name	age	city
1	John	26	New York
2	Sarah	31	Los Angeles
3	Alex	29	Chicago
4	Mary	36	Houston
5	Peter	28	San Francisco
6	John	26	New York
7	Alex	29	Chicago

Updated table after adding 1 year to age

The where clause allows selecting only those rows that satisfy a specified predicate:

```
1 SELECT name, age FROM sql_practice
2 WHERE age >= 30;
```

Relational Algebra:

$$\pi_{name,age}(\sigma_{age\geq 30}(sql\_practice)) \tag{1}$$

Resulting Table:

name	age
Sarah	30
Mary	35

Table after selecting partial attributes

Two sample tables for practices:

customer_id	country
1	USA
2	Japan
3	France
4	USA

Table 1: Table customers

order_id	ountry
100	USA
101	Japan
102	Italy
103	USA

Table orders

```
1    SELECT customers.customer_id, orders.order_id
2    FROM customers, orders
3    WHERE customers.country = orders.country;
```

This query will join the two tables on the column ‘country‘, and return the columns ‘customer_id‘ and ‘order_id‘ from each table where the values in the ‘country‘ column match.
The result of the query would be:

customer_id	order_id
1	100
2	101
4	100

the corresponding relational algebra expression for the given SQL query is:

$$\pi_{\text{customers.customer\_id,orders.order\_id}}\left(\sigma_{\text{customers.country}=\text{orders.country}}\left(\text{customers} \times \text{orders}\right)\right)$$

For-loop form of the multiple relation queries:

```sql
SELECT employees.name
FROM employees, departments, salaries
WHERE employees.emp_id = salaries.employee_id
AND salaries.department_id = departments.dept_id
AND salaries.salary > 55000
AND departments.name IN ('Sales', 'Marketing');
```

```
for each employee in employees:
    for each salary in salaries:
        for each department in departments:
            if employee.emp_id == salary.employee_id
            and salary.department_id == department.dept_id
            and salary.salary > 55000
            and department.name in ('Sales', 'Marketing'):
                ADD (employee.name) into the result relation
```

Break down queries on multiple relations step-by-step:

customers × orders

customers.customer_id	customers.country	orders.order_id	orders.country
1	USA	100	USA
1	USA	101	Japan
1	USA	102	Italy
1	USA	103	USA
2	Japan	100	USA
2	Japan	101	Japan
2	Japan	102	Italy
2	Japan	103	USA
3	France	100	USA
3	France	101	Japan
3	France	102	Italy
3	France	103	USA
4	USA	100	USA
4	USA	101	Japan
4	USA	102	Italy
4	USA	103	USA

Cartesian product of customers and orders tables

$$\sigma_{\text{customers.country}=\text{orders.country}}(\cdot)$$

customers.customer_id	customers.country	orders.order_id	orders.country
1	USA	100	USA
1	USA	103	USA
4	USA	100	USA
4	USA	103	USA

Filtered table with matching country column values

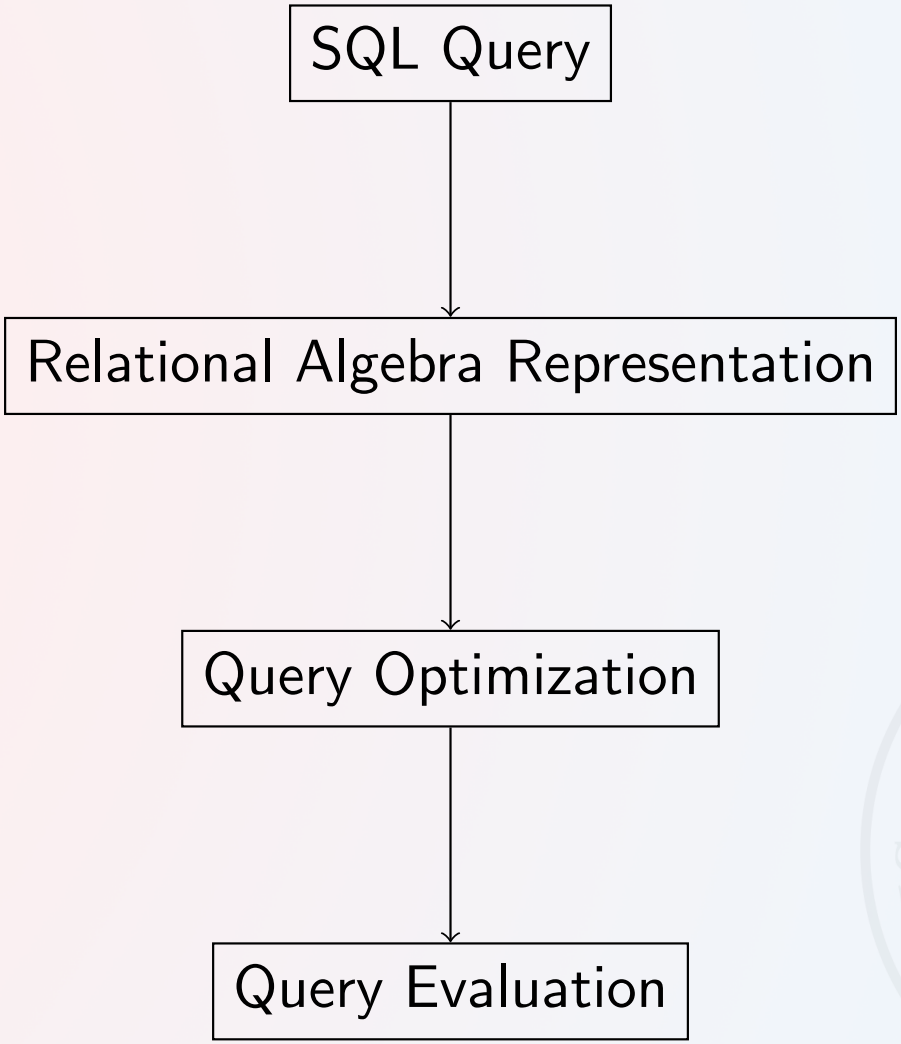
Be careful with WHERE clause. If you omit it, it could output a huge relation.

$$\pi_{\text{customers.customer\_id,orders.order\_id}}(\cdot)$$

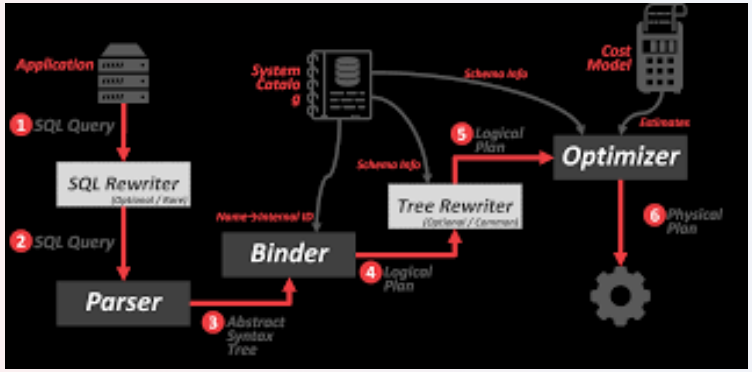
customers.customer_id	orders.order_id
1	100
1	103
4	100
4	103

Projected table with customer_id and order_id columns

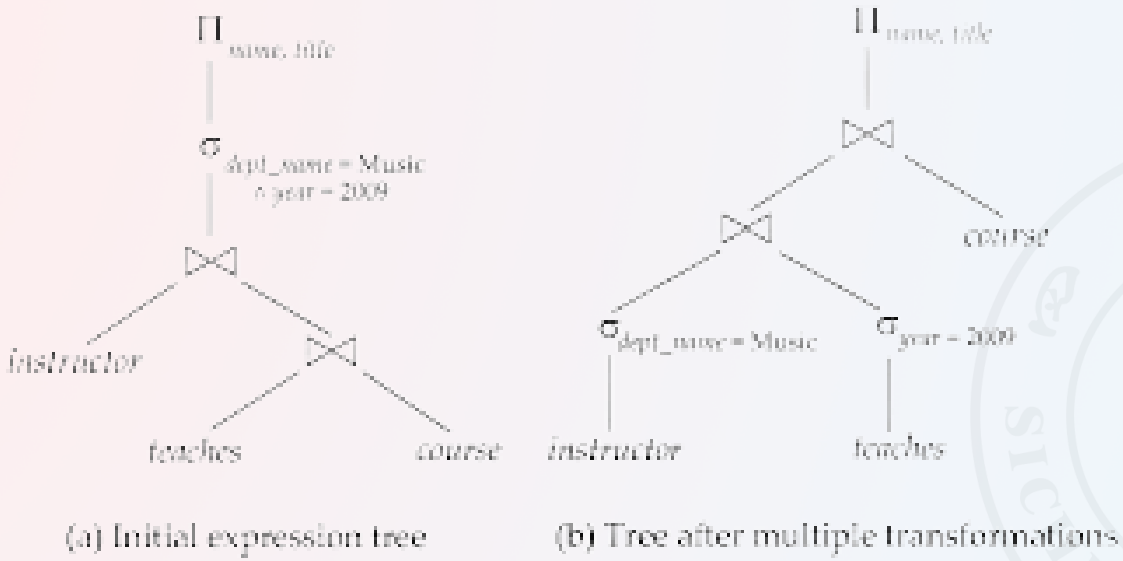
SQL queries and relational algebra



Chapter 15, 16



Query optimization



Tree Rewriter

目录

```sql
SELECT employee_id, first_name, last_name, salary AS wage
FROM employees;
```

This will produce a result set with the same columns as the "employees" table, except that the "salary" column will be renamed to "wage".

```sql
SELECT * FROM employees AS staff;
```

After executing this query, the table "employees" will be renamed to "staff". Any subsequent SQL queries that reference the table will need to use the new name "staff". "*" means all attributes.
In most cases, both TO and AS can be used to rename columns or tables and they are equivalent.

```sql
SELECT CONCAT('Hello', ' ', 'World');
-- Output: 'Hello World'
```

```sql
SELECT SUBSTRING('Hello World', 7);
-- Output: 'World'
```

```sql
SELECT LENGTH('Hello World');
-- Output: 11
```

```sql
SELECT LOWER('Hello World');
-- Output: 'hello world'

SELECT UPPER('Hello World');
-- Output: 'HELLO WORLD'
```

```sql
SELECT * FROM employees WHERE name LIKE '%son%';
```

The % wildcard at **the beginning and end** of the pattern matches any string that contains the string "son" anywhere in the name. This query will return all employees whose names contain the string "son".

employee_id	name
1	John Johnson
2	Robert Robertson
3	Michael Michaels
4	Sarah Stevenson

employee_id	name
1	John Johnson
2	Robert Robertson
3	Sarah Stevenson

目录

Name	Age	Salary	Department
John	30	60000	Sales
Mary	25	50000	Marketing
David	28	60000	IT
Michael	32	70000	Finance
Sarah	27	50000	HR

```
1 SELECT * FROM employees
2 ORDER BY salary DESC, name ASC;
```

Name	Age	Salary	Department
Michael	32	70000	Finance
David	28	60000	IT
John	30	60000	Sales
Mary	25	50000	Marketing
Sarah	27	50000	HR

目录

A table for practicing:

EmployeeID	Name	Salary	HireDate
101	John	50000	2018-01-01
102	Mary	55000	2019-03-15
103	Bob	60000	2020-05-20
104	Alice	65000	2017-12-31
105	Tom	55000	2021-02-28
106	Jane	55000	2018-07-01

Example employees table

```
1 SELECT * FROM employees
2 WHERE HireDate > '2019-01-01';
```

EmployeeID	Name	Salary	HireDate
3	Mary	55000	2019-03-15
4	Bob	60000	2020-05-20
5	Tom	55000	2021-02-28

Employees hired after January 1st, 2019

```
1 SELECT * FROM employees
2 WHERE Salary between 50000 and 55000;
```

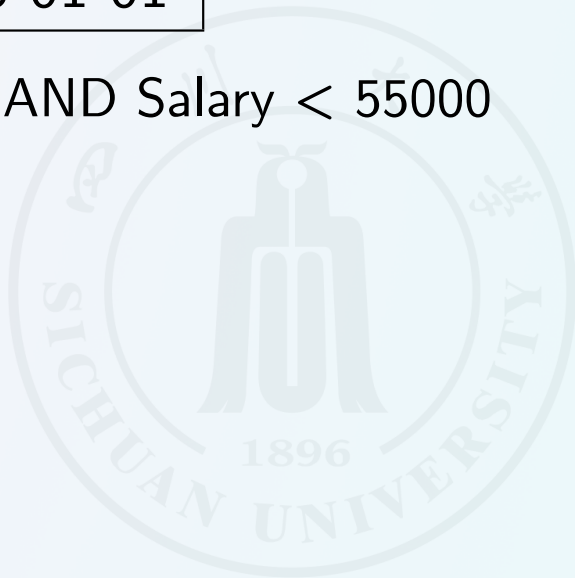
less than or equal to and greater than or equal to

EmployeeID	Name	Salary	HireDate
101	John	50000	2018-01-01
102	Mary	55000	2019-03-15
105	Tom	55000	2021-02-28
106	Jane	55000	2018-07-01

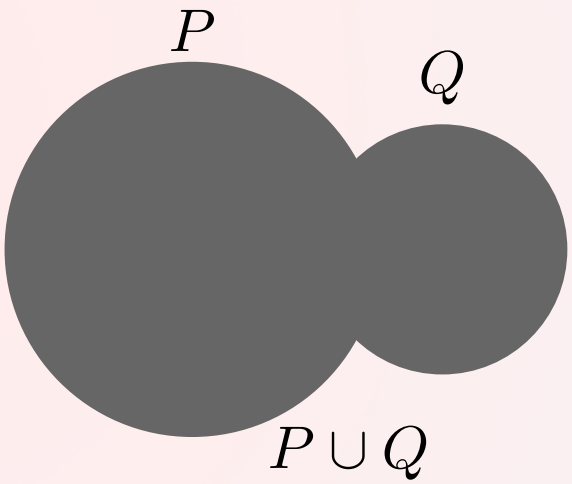
```sql
SELECT * FROM employees
WHERE HireDate < '2019-01-01' AND Salary < 55000;
```

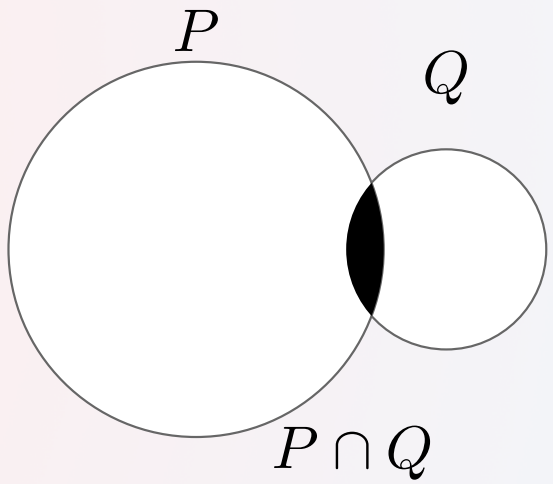
EmployeeID	Name	Salary	HireDate
101	John	50000	2018-01-01

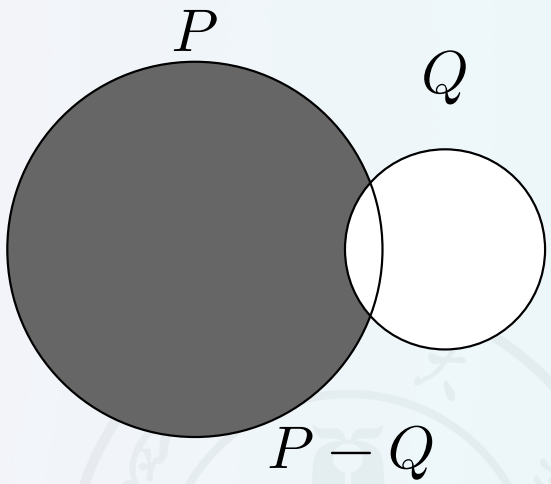
Example employees table with HireDate < '2019-01-01' AND Salary < 55000



The SQL operations union, intersect and except operate on relations and correspond to the relation algebra $\cup$, $\cap$ and $-$.







Tables for practicing:

EmployeeID	Name	Salary	Department
101	John	50000	Sales
102	Mary	55000	Marketing
103	Bob	60000	Engineering
104	Alice	65000	Sales
105	Tom	55000	Marketing
106	Jane	55000	Engineering

Table 1

EmployeeID	Name	Salary	Department
107	Jack	50000	HR
108	Sarah	45000	Engineering
109	David	70000	Sales
110	Lisa	55000	Marketing
111	Mike	60000	Engineering
112	Emily	60000	HR

Table 2

```
1 (SELECT * FROM table1
2 WHERE Department = 'Sales')
3 UNION ALL
4 (SELECT * FROM table2
5 WHERE Department = 'Engineering');
```

EmployeeID	Name	Salary	Department
101	John	50000	Sales
104	Alice	65000	Sales
108	Sarah	45000	Engineering
111	Mike	60000	Engineering

```sql
(SELECT Department FROM table1)
INTERSECT
(SELECT Department FROM table2);
```

Department
Sales
Marketing
Engineering

Intersect table (**Autimatical eliminate duplicates**)

```sql
(SELECT Department FROM table2)
EXCEPT ALL
(SELECT Department FROM table1);
```

Department
HR

ALL is used to keep duplicates

Null Values in SQL

Null values in SQL are special values that indicate the absence of a value:

- Null values are not equal to any other value, including other null values. This means that comparisons (<, =, >) involving null values always return **unknown**, even if the two values being compared are both null.
- Null values can appear in any column of a table, and they are typically used to represent missing or unknown data.
- Null values can make some SQL operations more complex. For example, the result of a calculation(+, -, * /) that involves a null value is also null, which can lead to unexpected behavior.
- Null values can be handled using special SQL keywords, such as IS NULL and IS NOT NULL. These keywords allow you to check whether a value is null or not.

SQL Unknown

In addition to null values, SQL also has the concept of "unknown" values ("not (1 < null)").
SQL uses a three-valued logic system that includes the values of true, false, and unknown. When performing boolean operations involving unknown values, the following rules apply:

- AND: The result is unknown if either operand is unknown, false if either operand is false, and true only if both operands are true.
- OR: The result is unknown if either operand is unknown, true if either operand is true, and false only if both operands are false.
- NOT: The result is unknown if the operand is unknown, false if the operand is true, and true if the operand is false.

Consider a table "mytable" with the following **columns (attributes)**: "id" (integer), "name" (string), "age" (integer), and "status" (string). Some of the **rows (tuples)** in this table contain null or unknown values.

```
1 SELECT * FROM mytable WHERE age IS NULL;
```

This query returns all rows in "mytable" where the value of the column "age" is null.

```
1  SELECT * FROM mytable WHERE age IS UNKNOWN;
```

This query returns all rows in "mytable" where the value of the column "age" is unknown.

In SQL, the SELECT DISTINCT statement is used to retrieve only unique values from a column. When dealing with NULL values(**tricky**).

ID	Value
1	A
2	B
3	C
4	NULL
5	NULL

```
1 SELECT DISTINCT Value FROM table
```

Return: **"A", "B", "C", and NULL**.
NULL = NULL is unknown

Suppose we have the following table:

ID	Score
1	85
2	90
3	75
4	80
5	95

Function	Result
COUNT(*)	5
AVG(Score)	85
SUM(Score)	425
MAX(Score)	95
MIN(Score)	75

```sql
SELECT Name, AVG(Score) AS avgscore
FROM Students WHERE XXX;
```

```sql
SELECT COUNT(DISTINCT ID)
FROM Students WHERE XXX;
```

SQL does not allow the use of DISTINCT with COUNT(*)

目录

Sales:

- product_id: the ID of the product being sold
- customer_id: the ID of the customer making the purchase
- sale_date: the date of the sale
- sale_amount: the amount of the sale

What is the total sales amount for each product?

```sql
SELECT product_id, SUM(sale_amount) as total_sales FROM sales
GROUP BY product_id;
```

How many sales did each customer make?

```sql
SELECT customer_id, COUNT(*) as num_sales FROM sales
GROUP BY customer_id;
```

Employee:

- employee_id: the ID of the employee
- department_id: the ID of the department the employee works in
- salary: the salary of the employee

```
1 SELECT department_id, employee_id, AVG(salary) as avg_salary
2 FROM employees
3 GROUP BY department_id;
4
5 ERROR 1055 (42000): Expression #2 of SELECT list is not in GROUP BY
       clause and contains nonaggregated column 'employees.
      employee_id' which is not functionally dependent on columns in
      GROUP BY clause; this is incompatible with sql_mode=
      only_full_group_by
```

This error occurs because we are trying to include the employee_id column in the result set, but **it is not included in the GROUP BY clause**.

OrderID	CustomerID	Total
1	100	50.00
2	200	100.00
3	100	75.00
4	300	25.00
5	200	125.00
6	100	90.00

Orders

```
1 SELECT CustomerID, SUM(Total) AS OrderTotal
2 FROM orders
3 GROUP BY CustomerID
4 HAVING OrderTotal > 100
```

CustomerID	OrderTotal
100	215.00
200	225.00

Consider the following SQL statement:

```
1 SELECT column1, column2, aggregate_function(column3)
2 FROM table
3 WHERE condition
4 GROUP BY column1, column2
5 HAVING condition2
```

This statement performs the following steps:

1. **FROM**: Retrieve the data from the specified table.
2. **WHERE**: Filter the data based on the specified condition.
3. **GROUP BY**: Group the filtered data by the specified columns.
4. **HAVING**: Filter the grouped data based on the specified condition.
5. **SELECT**: Select the specified columns and apply the specified aggregate function to column3 for each group.

目录

A null value represents the absence of a value in a database table. When performing aggregation functions, null values are typically ignored. Exception:

```
1 SELECT COUNT(column1) FROM table
```

This query will count the number of values in column1 (include null values).

```
1 SELECT AVG(column1) FROM table
```

If column1 contains a null and/or unknown value, it will be treated as a null value and excluded from the average calculation.

目录

Consider the following SQL query:

```
1 SELECT A1, A2, SUM(A3)
2 FROM r1, r2, \cdots, rm
3 WHERE P
4 GROUP BY A1, A2
5 HAVING COUNT(A4) > 2
```

Relational algebra:

$$R1 \leftarrow \sigma_P(r1 \times r2 \times \cdots \times rm)$$

$$R2 \leftarrow ({}_{A1,A2}\gamma_{SUM(A3)asSumA3,COUNT(A4)asCountA4}(R1))$$

$$\Pi_{A1,A2,SUM(A3)}\left(\sigma_{COUNT(A4)>2}R2\right)$$

目录

One common application of SQL is to query whether a specific value or data item exists(**Set Membership**)
Customers(CustomerID, CustomerName, Address)
Execute:

```
1 SELECT * FROM orders
2 WHERE CustomerID IN (SELECT CustomerID FROM customers);
```

- This query will return all customer IDs that exist in the "orders" table and also exist in the "customers" table.
- If there are no such orders, the query will return an empty result set.
- NOT IN return all IDs that do not exist in the "orders" table

```
1    SELECT DISTINCT T.name
2    FROM instructor AS T, instructor AS S
3    WHERE T.salary > S.salary AND S.depLname = 'Biology';
```

We can use SOME (at least one) to achieve the same task.

```
1    SELECT name
2    FROM instructor AS T
3    WHERE salary > SOME (SELECT salary FROM instructor WHERE
         depLname = 'Biology';
```

The 'some' keyword can also be used with other comparison operators, such as '<', '<=', '>', '>=', and '< >'.

name	salary	department
Alice	50000	HR
Bob	60000	Sales
Charlie	40000	IT
Dave	70000	Sales
Eve	80000	HR

```sql
SELECT name, salary
FROM employees
WHERE salary >= SOME (SELECT salary FROM employees WHERE department
      = 'Sales');
```

name	salary
Bob	60000
Dave	70000
Eve	80000

```sql
SELECT name, salary
FROM employees
WHERE salary >= ALL (SELECT salary FROM employees WHERE department
    = 'Sales');
```

name	salary
Dave	70000
Eve	80000

```sql
SELECT name, salary
FROM employees
WHERE salary <> SOME (SELECT salary FROM employees WHERE department
    = 'Sales');
```

name	salary
Alice	50000
Charlie	40000
Dave	70000
Eve	80000

```sql
SELECT name, salary
FROM employees
WHERE salary NOT IN (SELECT salary FROM employees WHERE department
    = 'Sales');
```

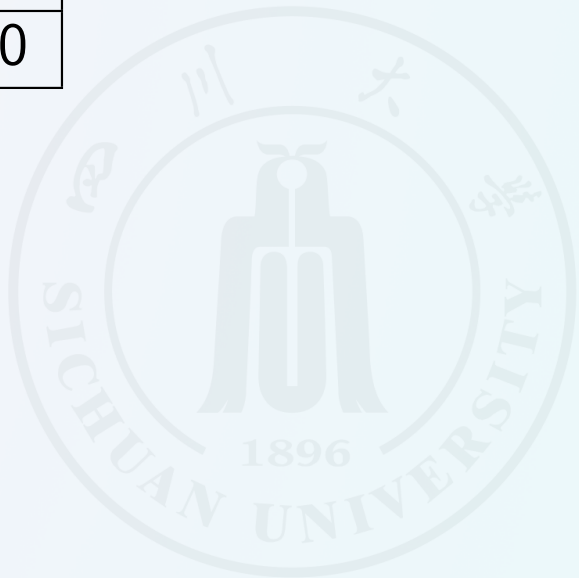
name	salary
Alice	50000
Charlie	40000
Eve	80000

```sql
SELECT department, AVG(salary) as avg_salary
FROM employees
WHERE salary >= SOME (SELECT salary FROM employees WHERE department
      = 'Sales')
GROUP BY department
HAVING COUNT(*) >= 2;
```

- WHERE salary >= SOME (SELECT salary FROM employees WHERE department = 'Sales'): filters the employees by selecting only those whose salary is greater than or equal to some value in the set of salaries for employees in the Sales department.
- GROUP BY department: groups the result set by the department column.
- HAVING COUNT(*) >= 2: filters the grouped results by selecting only those departments that have at least two employees with a salary greater than or equal to the maximum salary of the Sales department.

id	name	department	salary
1	Alice	Sales	50000
2	Bob	Sales	60000
3	Charlie	Marketing	40000
4	David	Marketing	45000
5	Eve	Marketing	55000
6	Frank	HR	35000
7	George	HR	40000

department	avg_salary
Sales	55000.00



目录

order_id	customer_id	order_date
1	101	2022-01-01
2	102	2022-01-02
3	103	2022-01-03

Orders table

customer_id	name
101	Alice
102	Bob
103	Charlie
104	David

Customers table

```
1 SELECT customer_id, name
2 FROM customers
3 WHERE NOT EXISTS (SELECT * FROM orders WHERE orders.customer_id =
      customers.customer_id);
```

customer_id	name
104	David

Customers with no orders

```
1 SELECT customer_id, name
2 FROM customers
3 WHERE EXISTS (SELECT * FROM orders WHERE orders.customer_id =
      customers.customer_id);
```

customer_id	name
101	Alice
102	Bob
103	Charlie

Customers with orders

InstructorID	Name	Department	Salary
44547	Smith	Computer Science	95000
44541	Bill	Electrical	55000
47778	Sam	Humanities	44000
48147	Erik	Mechanical	80000
411547	Melisa	Information Technology	65000
48898	Jena	Civil	50000

Instructor

Department	Budget
Computer Science	100000
Electrical	80000
Humanities	50000
Mechanical	40000
Information Technology	90000
Civil	60000

Department

Find all professors whose salary is greater than the average budget of all the departments:

```
1 SELECT I.ID, I.NAME, I.DEPARTMENT, I.SALARY FROM
2 (SELECT avg(BUDGET) AS averageBudget FROM DEPARTMENT) AS BUDGET,
      Instructor AS I
3 WHERE I.SALARY > BUDGET.averageBudget;
```

ID	Name	Department	Salary
44547	Smith	Computer Science	95000
48147	Erik	Mechanical	80000

Output

Syntax

```
1 SELECT <Column Name>
2 FROM <Reference Table Name>,
      LATERAL <Inner Subquery>
```

Description

The LATERAL keyword allows you to reference columns from a previous table. This can be useful for calculating values or filtering data based on a previous table.

With Clause 目录

WITH Clause in SQL

Syntax

```
1 WITH alias AS (
2     SELECT ...
3     FROM table
4     WHERE ...
5 )
6 SELECT ...
7 FROM alias
```

Description

The `WITH` clause allows you to define a temporary result set that can be referenced in the main query. This can simplify complex queries and improve readability.

Scalar Subqueries in SQL

Syntax

```
1 SELECT column_a, (
2     SELECT MAX(column_b)
3     FROM table_b
4     WHERE table_b.col = table_a.
          col
5 ) AS max_b
6 FROM table_a
```

Description

A scalar subquery is a subquery that returns a single value, which can be used as an expression in the main query. This can be useful for calculating values based on data from another table or column. In this example, we are using a scalar subquery to calculate the maximum value of `column_b` for each row in `table_a`, based on a matching value in `table_b`.

```sql
SELECT MAX(salary) FROM employees;
```

```sql
SELECT department_id, COUNT(*)
FROM employees
GROUP BY department_id;
```

```sql
SELECT name
FROM departments
WHERE budget = (SELECT MIN(budget) FROM departments);
```

```sql
SELECT AVG(salary)
FROM employees
WHERE department_id = (SELECT id FROM departments WHERE name = 'HR'
    );
```

Semijoins

A semijoin between two tables returns only the rows from the first table where there is a match in the second table. The SQL syntax for a semijoin is:

```sql
SELECT *
FROM table_a
WHERE EXISTS (
    SELECT *
    FROM table_b
    WHERE table_a.col = table_b.col
)
```

The mathematical expression for a semijoin is:

$$A \text{ semijoin } B = \pi_A(A \bowtie B)$$

Antijoins

An antijoin between two tables returns only the rows from the first table where there is no match in the second table. The SQL syntax for an antijoin is:

```
SELECT *
FROM table_a
WHERE NOT EXISTS (
    SELECT *
    FROM table_b
    WHERE table_a.col = table_b.col
)
```

The mathematical expression for an antijoin is:

$$A \text{ antijoin } B = \pi_A(A - A \bowtie B)$$

To **retrieve** all employees with a salary greater than 50000:

```
1 SELECT * FROM employees WHERE salary > 50000;
```

To **update** the salary of an employee with ID 1234 to 60000:

```
1 UPDATE employees SET salary = 60000 WHERE id = 1234;
```

To **delete** all employees with a salary less than 40000:

```
1 DELETE FROM employees WHERE salary < 40000;
```

To **add** a new employee with ID 5678, name "John Doe", and salary 45000:

```
1 INSERT INTO employees (id, name, salary) VALUES (5678, 'John Doe',
      45000);
```

Deletion 目录

```
1 DELETE FROM table\_name WHERE condition;
```

1. Note that a DELETE statement operates on **only one** relation.
2. The database engine checks the specified condition to determine which rows should be deleted.
3. For each row(tuple，元组) that meets the condition, the database engine removes the row from the table.
4. If any foreign key constraints exist that reference the deleted rows, the database engine will take the appropriate action to maintain referential integrity.
5. The database engine updates any indexes and triggers that may be affected by the delete operation.

1. **CASCADE**: This option automatically deletes the rows in the referencing table that correspond to the deleted rows in the referenced table.
2. **SET NULL**: This option sets the values in the referencing table's foreign key column to NULL for any rows that correspond to the deleted rows in the referenced table.
3. **RESTRICT**: The DELETE operation will fail if there are any rows in table B that reference the deleted row in table A.
4. **NO ACTION**: This option raises an error if there are any rows in the referencing table that reference the deleted rows in the referenced table.

ON DELETE

```
1 ALTER TABLE orders ADD FOREIGN KEY (customer_id) REFERENCES
      customers(id) ON DELETE CASCADE;
```

Insertion 目录

id	first_name	last_name	age
1	John	Smith	25
2	Jane	Doe	30
3	Bob	Johnson	45

Example Table: `users`

```
1 INSERT INTO users (id, first_name, last_name, age) VALUES (4, '
      Sarah', 'Johnson', 28);
```

Insert a new row into the `users` table with an `id` of 4, a `first_name` of 'Sarah', a `last_name` of 'Johnson', and an age of 28.

id	first_name

Example Table: employees

```
1 INSERT INTO employees (id, first_name) SELECT id, first_name FROM
      people WHERE age > 30;
```

id	first_name
3	Bob

Example Table: employees

```sql
INSERT INTO users (last_name) VALUES ('Alice');
```

id	first_name	last_name	age
1	John	Smith	25
2	Jane	Doe	30
3	Bob	Johnson	45
NULL	**NULL**	Alice	**NULL**

Example Table: `users`

If no **default value** is specified, the attribute will be assigned a **NULL** value.

A bulk loader is a utility provided by the database system that can load data from a file into a table in the database (**faster than insert**).

```
1 LOAD DATA INFILE '/path/to/employee_data.csv'
2 INTO TABLE employees
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY '"'
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS;
```

- LOAD DATA INFILE: load the data from the file
- FIELDS TERMINATED BY: clause specifies that the fields in the CSV file are separated by
- ENCLOSED BY: specifies that fields that contain commas should be enclosed in double quotes
- LINES TERMINATED BY: specifies that each row of data is terminated by a newline character
- IGNORE 1 ROWS: skip the first row of the file

Updates 目录

```sql
UPDATE employees
SET salary = 60000
WHERE id = 100;
```

updates the salary of the employee with ID 100 to 60000 in the employees table.

```sql
UPDATE orders
SET status = 'Canceled'
WHERE customer_id = 123 AND status = 'Pending';
```

cancels all orders with a customer_id of 123 and a status of "Pending" in the orders table.

```sql
UPDATE employees
SET salary = salary * 1.1
WHERE department_id IN (
  SELECT id
  FROM departments
  WHERE name = 'Sales'
);
```

increases the salary of all employees in the "Sales" department by 10%. It uses a subquery to find the IDs of all departments named "Sales".

```sql
UPDATE table_name SET column1 = CASE
WHEN condition1 THEN value1
WHEN condition2 THEN value2
 ...
 ELSE default_value
 END
WHERE condition;
```

```sql
UPDATE employees SET salary = CASE
 WHEN department = 'Sales' THEN salary * 1.1
 WHEN department = 'Marketing' THEN salary * 1.2
 ELSE salary
 END
WHERE id IN (1, 2, 3);
```

This statement updates the salaries of employees with IDs 1, 2, and 3 in the employees table, based on their department.

id	name	gpa	status
1	Alice	3.5	NULL
2	Bob	3.0	NULL
3	Charlie	2.8	NULL
4	Dave	3.9	NULL
5	Emma	2.5	NULL

Students

```
1 UPDATE students
2 SET status = CASE WHEN gpa > (SELECT AVG(gpa) FROM students) THEN '
      good'
3 ELSE 'needs improvement' END;
```

id	name	gpa	status
1	Alice	3.5	good
2	Bob	3.0	good
3	Charlie	2.8	needs improvement
4	Dave	3.9	good
5	Emma	2.5	needs improvement

Practice complex queries on flight delay dataset. (在航班延误数据集上练习各种 SQL 语言，输出你想要的结果，写五个较为复杂的航班延误分析，譬如说，想要得到各航班公司在不同机场上的航班延误占比，SQL 可以通过从 CSV 文件里构建数据库)

Chamberlin, Donald D., Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. "SEQUEL 2: A unified approach to data definition, manipulation, and control." IBM Journal of Research and Development 20, no. 6 (1976): 560-575.

Thanks
End of Chapter 3