清华大学数据库技术与应用

SQL III

授课教师：计算机系王健楠

授课学期：2026年（春季）

课程大纲

聚合查询

- 简单聚合
- 分组（Group By）
- 分组筛选（Having）

结果复用

- Table / View / CTE

子查询

- 在 FROM 子句中
- 在 WHERE 子句中

简单聚合

```
SELECT agg(column)
FROM   <table name>
WHERE  <conditions>
```

agg = COUNT, SUM, AVG, MAX, MIN, etc.

除 COUNT 外，其他聚合函数均作用于单个属性

示例

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	4

```
SELECT COUNT(*) FROM Student
```

5

```
SELECT SUM(gpa) FROM Student
```

17

```
SELECT AVG(gpa) FROM Student
```

3.4

```
SELECT MIN(gpa) FROM Student
```

3

```
SELECT MAX(gpa) FROM Student
```

4

示例

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	4

```
SELECT COUNT(DISTINCT gpa) FROM Student
```

2

```
SELECT SUM(DISTINCT gpa) FROM Student
```

7

```
SELECT AVG(gpa) FROM Student WHERE
gender = 'F'
```

3.5

为什么需要 GROUP BY

- 如何获取每种性别的平均 GPA?

```
SELECT AVG(gpa) FROM Student WHERE gender = 'M'
```

```
SELECT AVG(gpa) FROM Student WHERE gender = 'F'
```

- 如何获取每个年龄段的平均 GPA?

```
SELECT AVG(gpa) FROM Student WHERE age = 18
```

```
SELECT AVG(gpa) FROM Student WHERE age = 19
```

```
SELECT AVG(gpa) FROM Student WHERE age = 20
```

•
•
•

分组与聚合

```
SELECT agg(column)
FROM   <table name>
WHERE  <conditions>
GROUP BY <columns>
```

- 如何计算每个性别的平均 GPA?

```
SELECT AVG(gpa) FROM Student GROUP BY gender
```

- 如何计算每个年龄的平均 GPA?

```
SELECT AVG(gpa) FROM Student GROUP BY age
```

分组与聚合

以下查询是如何执行的?

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 2.5
GROUP BY gender
```

查询的语义

- 第 1 步：计算 FROM 和 WHERE 子句
- 第 2 步：按 GROUP BY 属性分组
- 第 3 步：计算 SELECT 子句（分组属性和聚合值）

步骤 1：执行 FROM 与 WHERE 子句

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 2.5
GROUP BY gender
```

name	gender	gpa
Bob	M	2
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	3

name	gender	gpa
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	3

步骤 2：按 GROUP BY 属性分组

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 2.5
GROUP BY gender
```

name	gender	gpa
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	3

gender	name	gpa
M	Mike	3
	Tom	3
F	Alice	3
	Mary	4

步骤 3：计算 SELECT 子句

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 2.5
GROUP BY gender
```

gender	name	gpa
M	Mike	3
	Tom	3
F	Alice	3
	Mary	4

gender	AVG(gpa)
M	3
F	3.5

练习1

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

gender	AVG(gpa)
F	4

(A)

VS

gender	AVG(gpa)
F	4
M	NULL

(B)

练习：空组

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

name	gender	gpa
Alice	F	4
Mary	F	4

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

练习：空组

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

name	gender	gpa
Alice	F	4
Mary	F	4

gender	name	gpa
F	Alice	4
	Mary	4

练习：空组

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

name	gender	gpa
Alice	F	4
Mary	F	4

gender	name	gpa
F	Alice	4
	Mary	4

gender	AVG(gpa)
F	4

练习2

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa), name
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

gender	AVG(gpa)	name
F	4	Alice

(A)

VS

gender	AVG(gpa)	name
F	4	Mary

(B)

练习：非法 SELECT

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

name	gender	gpa
Alice	F	4
Mary	F	4

```
SELECT gender, AVG(gpa), name
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

练习：非法 SELECT

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa), name
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

name	gender	gpa
Alice	F	4
Mary	F	4

gender	name	gpa
F	Alice	4
	Mary	4

练习：非法 SELECT

SELECT 中的每个属性，必须是 GROUP BY 的分组属性，或是聚合函数

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa), name
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

name	gender	gpa
Alice	F	4
Mary	F	4

gender	name	gpa
F	Alice	4
	Mary	4

gender	AVG(gpa)	name
F	4	???

HAVING 子句

用于指定感兴趣的分组条件

```
SELECT agg(column)
FROM   <table name>
WHERE  <conditions>
GROUP BY <columns>
HAVING <columns>
```

HAVING 子句

与前例相同，但要求每组学生数超过 10

```
SELECT AVG(gpa), gender
FROM Student
WHERE gpa > 2.5
GROUP BY gender
HAVING COUNT(*) > 10
```

HAVING 子句包含对聚合结果的筛选条件。

执行顺序

清華大學 Tsinghua University

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

- 对 FROM 子句中的各表做笛卡尔积
- 删除不满足 WHERE 条件的行
- 按 GROUP BY 子句将记录分组
- 删除不满足 HAVING 条件的分组
- 为每组生成一行，并删除 SELECT 子句中未列出的列

练习

StudentInfo

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa)
FROM StudentInfo
WHERE gpa > 2.5
GROUP BY gender
HAVING COUNT(*) > 2
```

```
SELECT gender, AVG(gpa)
FROM StudentInfo
WHERE gpa > 2.5
GROUP BY gender
HAVING SUM(gpa) < 9
```

gender	AVG(gpa)
M	3

(A)

gender	AVG(gpa)
F	4

(B)

gender	AVG(gpa)
M	3
F	4

(C)

课程大纲

聚合查询

- 简单聚合
- 分组（Group By）
- 分组筛选（Having）

结果复用

- Table / View / CTE

子查询

- 在 FROM 子句中
- 在 WHERE 子句中

结果复用：Table / View / CTE

- 有时需要将查询结果保存或命名，以便在后续查询中复用
- 常用三种策略：
  - Table — 将结果写入磁盘，成为独立的永久表
  - View — 保存查询定义，每次使用时重新计算
  - CTE — 在当前语句中定义临时命名结果
- 三种策略各有适用场景，按需选择

Table：保存一份副本

- 将查询结果一次性写入磁盘，创建一张新表
- 数据不随原表更新而同步——是数据的快照副本
- 适合需要独立保留某个时刻结果的场景

```
CREATE TABLE CustomerBalances AS (SELECT O.customerID, SUM(A.balance) AS sumBalance
  FROM Account A, Owns O
  WHERE O.accNumber = A.accNumber
  GROUP BY O.customerID);
```

View：保存查询定义

- 只保存查询的定义，不存储数据本身
- 每次查询 View 时重新执行，结果始终反映最新数据
- 适合需要复用复杂查询、但始终需要最新结果的场景

```
CREATE VIEW CustomerBalances_view AS
  SELECT O.customerID, SUM(A.balance) AS sumBalance
   FROM Account A, Owns O
   WHERE O.accNumber = A.accNumber
   GROUP BY O.customerID;
```

CTE：语句内的临时结果

- 用 WITH 子句在当前 SQL 语句中定义临时命名结果
- 只在当前语句中可见，语句执行完毕后不保留
- 适合需要在同一语句中多次引用中间结果、提升可读性的场景

```
WITH CustomerBalances AS
  (SELECT O.customerID, SUM(A.balance) AS sumBalance
   FROM Account A, Owns O
   WHERE O.accNumber = A.accNumber
   GROUP BY O.customerID)

SELECT * FROM CustomerBalances;
```

用 CTE 复用中间结果

- CTE 的核心优势：同一中间结果可在语句中多次引用
- 示例：找出总余额最高的客户（CustomerBalances 被引用两次）

```
WITH CustomerBalances AS
  (SELECT O.customerID, SUM(A.balance) AS sumBalance
   FROM Account A, Owns O
   WHERE O.accNumber = A.accNumber
   GROUP BY O.customerID)

SELECT customerID FROM CustomerBalances
WHERE sumBalance = (SELECT MAX(sumBalance) FROM CustomerBalances);
```

什么时候用 Table / View / CTE?

- **Table** — 需要保留某时刻的快照；数据独立于原表；可频繁查询且无需每次重算
- **View** — 复杂查询需反复使用；始终要求最新数据；不想占用存储空间
- **CTE** — 仅在单条语句内使用；同一中间结果需引用多次；想让查询结构更清晰

课程大纲

聚合查询

- 简单聚合
- 分组（Group By）
- 分组筛选（Having）

结果复用

- Table / View / CTE

子查询

- 在 FROM 子句中
- 在 WHERE 子句中

子查询

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

- 子查询是嵌套在外层查询中的 SQL 查询
- 这种内外层查询结构称为嵌套查询

```
SELECT C.customerID, C.birthDate, C.income
FROM Customer C
WHERE C.customerID IN
    (
      SELECT O.customerID
      FROM Account A, Owns O
      WHERE A.accNumber = O.accNumber
          AND A.branchName = London'
    )
```

外层查询

内层查询

子查询

子查询可以出现在以下子句中

- FROM 子句
- WHERE 子句
- HAVING 子句

```
SELECT <columns>
FROM   <table name>
WHERE  <conditions>
GROUP BY <columns>
HAVING <columns>
```

FROM 子句中的子查询

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

- 有时需要先计算一个中间表，再在 SELECT-FROM-WHERE 中使用
- **谁是最富有的客户?**

```
SELECT firstName, lastName, MAX(sumBalance)
FROM (SELECT firstName, lastName, sum(balance) AS sumBalance
        FROM Customer C, Account A, Owns O
        WHERE C.customerID = O.customerID
                AND O.accNumber = A.accNumber
        GROUP BY C.customerID )
```

FROM 子句中的子查询

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

- 有时需要先计算一个中间表，再在 SELECT-FROM-WHERE 中使用
- **哪些客户的账户总余额为 0?**

```
SELECT firstName, lastName, sumBalance
FROM (SELECT firstName, lastName, sum(balance) AS sumBalance
        FROM Customer C, Account A, Owns O
        WHERE C.customerID = O.customerID
                AND O.accNumber = A.accNumber
        GROUP BY C.customerID) AS T
WHERE T. sumBalance = 0
```

FROM 子句中的子查询

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID$^{FK\text{-}Customer}$, accNumber$^{FK\text{-}Account}$}

- 有时需要先计算一个中间表，再在 SELECT-FROM-WHERE 中使用
- **哪些客户的账户总余额为 0?**

```
SELECT firstName, lastName, sum(balance) AS sumBalance
FROM Customer C, Account A, Owns O
WHERE C.customerID = O.customerID AND O.accNumber = A.accNumber
GROUP BY C.customerID
HAVING sumBalance = 0
```

原则：尽量避免使用嵌套查询

WHERE 子句中的子查询

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

- 子查询返回单个常量时，可与比较运算符连用
  - >, <, =, <>, >=, <=
- **查找收入高于平均值avg(income)的客户 ID**

```
SELECT C1.customerID
FROM Customer C1
WHERE C1.income > (SELECT avg(C2.income)
                   FROM Customer C2)
```

WHERE 子句中的子查询

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

子查询返回一个关系时，可用以下运算符

- IN
- NOT IN
- EXISTS
- NOT EXISTS
- ANY
- ALL

London支行的账户

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

查找在**London支行**有账户的客户 ID

```
SELECT C.customerID
FROM Customer C
WHERE C.customerID IN (SELECT O.customerID
                       FROM Account A, Owns O
                       WHERE A.accNumber = O.accNumber
                           AND A.branchName = 'London')
```

非London支行的账户

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID$^{FK\text{-}Customer}$, accNumber$^{FK\text{-}Account}$}

查找在**London支行**没有账户的客户 ID

```
SELECT C.customerID
FROM Customer C
WHERE C.customerID NOT IN (SELECT O.customerID
                           FROM Account A, Owns O
                           WHERE A.accNumber = O.accNumber
                             AND A.branchName = 'London')
```

非相关查询

- 前面的查询包含非相关（独立）子查询
  - 子查询不引用外层查询的任何属性

- 独立子查询可以在外层查询执行前先行求值
  - 且只需求值一次
    - 对外层查询的每一行，均可直接检查子查询结果
  - 代价 = 子查询执行一次的代价 + 扫描外层关系的代价

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

EXISTS：London支行账户

查找在**London支行**有账户的客户 ID

```
SELECT C.customerID
FROM Customer C
WHERE EXISTS ( SELECT *
                    FROM Account A, Owns O
                    WHERE C.customerID = O.customerID
                              AND A.accNumber = O.accNumber
                              AND A.branchName = 'London')
```

EXISTS 和 NOT EXISTS 用于判断关联子查询的结果是否为空

相关查询

前面的查询包含相关子查询

- 子查询引用了外层查询的属性
  - … WHERE C.customerID = O.customerID …
- 对外层查询的每一行，都需重新求值
  - 即对 Customer 表的每一行

相关查询通常效率较低

EXISTS：London支行账户（等效写法）

查找在**London支行**有账户的客户 ID

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

```
SELECT DISTINCT C.customerID
FROM Customer C, Account A, Owns O
WHERE C.customerID = O.customerID
    AND A.accNumber = O.accNumber
    AND A.branchName = 'London'
```

在所有支行都有账户

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}
Branch = {branchNumber, branchName, managerSIN[FK-Employee], budget}

查找在所有支行均有账户的客户 ID

SQ1 – 所有支行名称列表

SQ2 – 某客户拥有账户的支行名称列表

EXCEPT

若该客户在每个支行均有账户，则差集为空

在所有支行都有账户

综合以上思路，最终查询如下

```
SELECT C.customerID
FROM Customer C
WHERE NOT EXISTS ( (SELECT B.branchName
                    FROM Branch B)
                      EXCEPT
                   (SELECT A.branchName
                    FROM Account A, Owns O
                    WHERE O.customerID = C.customerID
                      AND O.accNumber = A.accNumber))
```

ANY：比某个 Joe 更富有

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID[FK-Customer], accNumber[FK-Account]}

查找收入高于某个名叫 Joe 的客户的所有客户 ID

结果中的客户收入，必须大于子查询结果中的至少一行

```
SELECT C.customerID
FROM Customer C
WHERE C.income > ANY
        (SELECT Joe.income
         FROM Customer Joe
         WHERE Joe.firstName = 'Joe')
```

结果中的客户收入，必须大于子查询结果中的至少一行

ALL：比所有Joe都富有

Customer = {customerID, firstName, lastName, income, birthDate}
Account = {accNumber, type, balance, branchName}
Owns = {customerID$^{FK\text{-}Customer}$, accNumber$^{FK\text{-}Account}$}

查找收入高于所有名叫 Joe 的客户的客户 ID

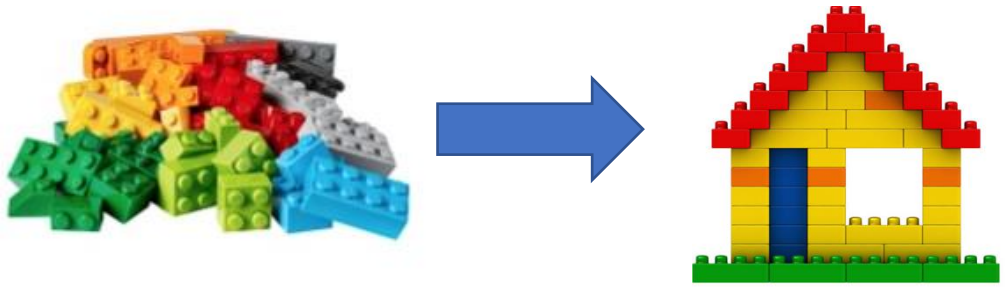
若不存在名叫 Joe 的客户，该查询将返回所有客户

```
SELECT C.customerID
FROM Customer C
WHERE C.income > ALL
        (SELECT Joe.income
         FROM Customer Joe
         WHERE Joe.firstName = 'Joe')
```

若不存在名叫 Joe 的客户，该查询将返回所有客户

总结

- 选择（Selection）
- 投影（Projection）
- 集合运算符（UNION、INTERSECT、EXCEPT）
- 连接（Joins）：INNER、OUTER
- 聚合（Aggregation）
- GROUP BY 分组
- HAVING 过滤
- ORDER BY 排序
- DISTINCT 去重
- 结果复用（Table / View / CTE）
- 子查询（Subqueries）



SQL 运算符可以像拼搭乐高积木一样自由组合