数据库系统概论

SQL语句

数据定义

操作对象	创建	删除	修改
数据库	CREATE DATABASE	DROP DATABASE	
模式	CREATE SCHEMA	DROP SCHEMA	
表	CREATE TABLE	DROP TABLE	ALTER TABLE
视图	CREATE VIEW	DROP VIEW	
索引	CREATE INDEX	DROP INDEX	ALTER INDEX

> 在MySQL中， `database` 就相当于 `schema` ，因此只需要创建 `database` 即可。

模式

模式是数据库中的一个命名空间，用于存放数据库对象，如表、视图、索引等。

```
CREATE SCHEMA <模式名> [AUTHORIZATION <用户名>];
DROP SCHEMA <模式名> <CASCADE|RESTRICT>;
```

- `<>` 表示必填内容， `[]` 表示可选内容。
- `CASCADE` （级联）表示删除模式时，该模式中的所有数据库对象会被全部删除；
- `RESTRICT` （限制）表示删除模式时，如果模式中有对象（如表、视图等）则不删除。

eg.

```
CREATE SCHEMA mydb;
DROP SCHEMA mydb CASCADE;
```

基本表

```
CREATE TABLE <表名> (
    <列名1> <数据类型1> [列级完整性约束条件1],
    <列名2> <数据类型2> [列级完整性约束条件2],
    ...
    [, <表级完整性约束条件>]
);
```

- 数据类型

数据类型	说明
CHAR(n)	定长为n的字符串
VARCHAR(n)	长度最大为n的字符串
INT	长整数，4字节
SMALLINT	短整数，2字节
BIGINT	长整数，8字节
FLOAT(n)	精度至少为n的浮点数
DECIMAL(p, q)	定点数，p为总位数，q为小数位数
BLOB	二进制大对象
BOOLEAN	布尔
DATE	日期，YYYY-MM-DD
TIME	时间，HH:MM:SS
TIMESTAMP	时间戳

- 列级完整性约束条件：
  - `NOT NULL`：非空；
  - `UNIQUE`：唯一；
  - `PRIMARY KEY`：主键；
- 表级完整性约束条件：
  - `FOREIGN KEY`：外键；
  - `PRIMARY KEY`：主键（可以有多个，此时只能定义在表级中）；

eg.

```
CREATE TABLE departments (
    dept_id INT AUTO_INCREMENT PRIMARY KEY,
    dept_name VARCHAR(100) NOT NULL UNIQUE,
    PRIMARY KEY (dept_id)
);

CREATE TABLE employees (
    emp_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(dept_id)
);
```

在该例子中， `employees.department_id` 是 `departments.dept_id` 的外键，即 `employees.department_id` 为 `departments.dept_id` 的引用。

对基本表的修改：

```
ALTER TABEL <表名>
[ADD [COLUMN] <新列名> <数据类型> [完整性约束条件]]
[ADD <表级完整性约束条件>]
[DROP [COLUMN] <列名> [CASCADE|RESTRICT]]
[DROP CONSTRAINT <完整性约束条件> [CASCADE|RESTRICT]]
[ALTER [COLUMN] <列名> <数据类型> [完整性约束条件]];
```

eg.

```
ALTER TABLE employees ADD COLUMN salary DECIMAL(10, 2);
ALTER TABLE employees ADD UNIQUE(emp_id);
ALTER TABLE employees DROP COLUMN salary CASCADE;
ALTER TABLE employees ALTER emp_name VARCHAR(200);
```

对基本表的删除：

```
DROP TABLE <表名> [CASCADE|RESTRICT];
```

索引

为基本表建立索引可以加速对数据的查询。

```
CREATE [UNIQUE] [CLUSTER] INDEX <索引名> ON <表名> (<列名1> [ASC|DESC] [, <列名2> [ASC|DESC], ...
```

- UNIQUE ： 索引的每一个索引值只对应唯一的数据记录
- CLUSTER ： 建立**聚簇索引**，要求表中数据按指定的聚簇属性值的升序/降序排序。
  - 每个基本表至多创建一个聚簇索引
  - 聚簇索引比一般索引的查询速度更快，但是插入、删除、更新的速度更慢。
  - 适用于很少对基表进行增删操作，但是经常查询的情况。
- 索引建立后会由系统自行维护，但是多个索引会减慢数据库更新的速度。

eg.

```
CREATE INDEX idx_emp_id ON employees(emp_id ASC);
```

可以修改索引的名字：

```
ALTER INDEX <索引名> RENAME TO <新索引名>;
```

删除索引：

```
DROP INDEX <索引名> ON <表名>;
```

数据查询

数据查询使用 SELECT 语句进行。

```
SELECT [ALL|DISTINCT] <目标列表达式> [别名] [, <目标列表达式> [别名], ...]
FROM <表名或视图名> [别名] [, <表名或视图名> [别名]]
[WHERE <条件表达式>]
[GROUP BY <列名1> [HAVING <条件表达式>]]
[ORDER BY <列名2> [ASC|DESC]];
```

- ALL ： 默认值，返回查询得到的全部结果；
- DISTINCT ： 去除结果中重复的行；
- <目标列表达式> ： 可以是列名、函数、常量等；

- 假设表中有 `id`， `name`， `age`，则目标列表达式可以为：

目标列表达式	描述
`id StuId`	查询表中的 `id` 列，在结果中把列重命名为 `StuId`
`'Year of Birth'`	查询结果中添加一列，列值为常量 `'Year of Birth'`
`2025-age BIRTHYEAR`	计算 2025 与 `age` 列的差值，并将结果列命名为 `BIRTHYEAR`
`*`	选择表中的所有列
标准函数	详见下方表格

SQL 标准函数

函数名	介绍
字符串函数	
`UPPER(str)`	将字符串 `str` 转换为大写。
`LOWER(str)`	将字符串 `str` 转换为小写。
`CONCAT(str1, str2, ...)`	连接（拼接）两个或多个字符串。 (部分方言使用 `
`LENGTH(str)`	返回字符串 `str` 的长度（字符数）。 (部分方言为 `LEN`)
`SUBSTRING(str FROM start [FOR length])`	从字符串 `str` 中提取子字符串，从 `start` 位置开始，可选长度 `length`。 (语法可能因方言略有不同，如 `SUBSTR(str, start, length)`)
`TRIM([LEADING | TRAILING | BOTH] [chars FROM] str)`	从字符串 `str` 的开头、结尾或两端移除指定的字符 `chars`（默认为空格）。
`REPLACE(str, from_str, to_str)`	在字符串 `str` 中，将所有出现的 `from_str` 替换为 `to_str`。
`POSITION(substring IN str)`	返回子字符串 `substring` 在字符串 `str` 中首次出现的位置（索引通常从 1 开始）。

函数名	介绍
数值函数	
ABS(num)	返回数值 num 的绝对值（非负值）。
ROUND(num [, decimals])	将数值 num 四舍五入到指定的小数位数 decimals （默认为 0）。
CEILING(num)	返回大于或等于数值 num 的最小整数值（向上取整）。(部分方言为 CEIL)
FLOOR(num)	返回小于或等于数值 num 的最大整数值（向下取整）。
MOD(num1, num2)	返回 num1 除以 num2 的余数。 (部分方言使用 % 操作符)
POWER(base, exponent)	返回 base 的 exponent 次幂。 (部分方言为 POW)
SQRT(num)	返回非负数值 num 的平方根。

- <条件表达式>： where子句的常用查询条件：

查询条件	谓词
比较	=，>，<，>=，<=，!=，<>（这两个都是不等于），!>，!<，NOT加前面的运算符
确定范围	[NOT] BETWEEN AND（闭区间）
确定集合	[NOT] IN，示例： where Sdept in ('CS', 'MA', 'IS')
字符匹配	[NOT] LIKE
空值	IS [NOT] NULL，**此处 IS 不可以用 = 代替**
多重条件	AND，OR，NOT

- 对于字符串匹配，有格式：
  - [NOT] LIKE '<匹配串> [ESCAPE '<转义字符>']
- 其中匹配串可以使用通配符：
  - % ： 匹配任意长度的字符串，可以为空；
  - _ ： 匹配单个字符；
  - ESCAPE ： 转义通配符；

    - 如， `a_b` 为a开头b结尾，中间一个字符； `a%b` 为a开头b结尾，中间任意字符； `%a%` 为包含a。
    - 转义字符可以任意定义，如 `LIKE 'a\%b' ESCAPE '\'` 。
- `ORDER BY` ：对查询结果按照指定列名的值进行升序/降序排序。
    - 空值被视为无穷大。
- `GROUP BY HAVING` ：对查询结果按照指定列名的值进行分组， `HAVING` 用于对分组后的结果进行筛选；
    - `HAVING` 中的条件表达式可以使用 **聚集函数**： （**where中不可以用聚集函数！！！**）

函数名	介绍
`COUNT(*)`	计算总行数
`COUNT([DISTINCT|ALL] <列名>)`	统计某列中**非空行**的行数
`SUM([DISTINCT|ALL] <列名>)`	计算一列**数值型**值的总和
`AVG([DISTINCT|ALL] <列名>)`	计算一列**数值型**值的平均值
`MAX([DISTINCT|ALL] <列名>)`	求一列值中的最大值
`MIN([DISTINCT|ALL] <列名>)`	求一列值中的最小值

> 其中 [DISTINCT|ALL] 默认为 `ALL` 。

eg.

```
SELECT COUNT(DISTINCT No) FROM SC;
SELECT SUM(score) FROM Stu WHERE no='201215121';
SELECT Cno, COUNT(Sno) FROM SC GROUP BY Cno; // 按课程分组，统计每门课程选课人数
SELECT Sno FROM SC GROUP BY Sno HAVING COUNT(*) > 3; // 统计选课超过3门的学生
```

注意：在 `GROUP BY` 之后，聚合函数的操作都将是针对**每个分组**内部进行的，而不是针对整个表。

连接查询（多表查询）

对涉及多个表的拆线呢操作称为 “连接查询”。

（非）等值连接

连接查询的 `where` 子句中用于连接两个表的条件称为 **连接条件** 或 **连接谓词** ，有格式：

```
[<表名1>.]<列名1> <比较运算符> [<表名2>.]<列名2>
[<表名1>.]<列名1> BETWEEN [<表名2>.]<列名2> AND [<表名2>.]<列名3>
```

其中比较运算符可以为：

- 等值连接： `=`
- 非等值连接： `>`， `<`， `>=`， `<=`， `!=`， `<>`

eg.

```
SELECT Stu.*, SC.* FROM Stu, SC WHERE Stu.Sno = SC.Sno;
```

获得的结果为：

Stu.Sno	Sname	Sdept	SC.Sno	Cno	Score
201215121	张三	CS	201215121	1	92
...	...	...	...	...	...

进行该连接时的过程称为 **嵌套循环连接**，即从 `Stu.Sno` 的第一行开始，从头遍历 `SC.Sno` 的每一行，找到相同的 `Sno` 值，然后将两行合并为结果表中的一行。若在 `SC.Sno` 上建立了索引，则不必遍历整个表，可以根据索引直接找到对应的行。

自然连接

即在等值连接的基础上，去掉重复的列。

eg.

```
SELECT Stu.Sno, Sname, Sdept, Cno, Score
FROM Stu, SC
WHERE Stu.Sno = SC.Sno;
```

注意，由于 `Sno` 在 `Stu` 和 `SC` 中都存在，因此引用时需要加上表名前缀，而其他列在两个表中是唯一的，因此不需要加表名前缀。

在一条where中可以同时完成连接和选择，如：

```
SELECT Stu.Sno, Sname
FROM Stu, SC
WHERE Stu.Sno = SC.Sno AND Score > 90;
```

自身连接

自身连接是指一个表与自身进行连接。

假设有Course表，其中Cpno为这门课的先修课：

Cno	Cname	Cpno
1	数据库	5
2	数学	
3	信息系统	1
4	操作系统	6
5	数据结构	7
6	数据处理	
7	C语言	6

现在要查询每一门课的先修课的先修课的课程号：

```
SELECT FIRST.Cno, SECOND.Cpno
FROM Course FIRST, Course SECOND
WHERE FIRST.Cno = SECOND.Cpno;
```

结果为：

Cno	Cpno
1	7
3	5
5	6

外连接

外连接分为 **左外连接** 和 **右外连接**：

- 左外连接会保留左表中的所有记录，右表中没有匹配的记录会以NULL填充。
- 右外连接同理。

语法为：

```
FROM <表名> <LEFT|RIGHT> OUTER JOIN <表名> ON (<连接条件>)
```

eg.

```
SELECT Stu.Sno, Sname, Sdept, Cno, Score
FROM Stu LEFT OUTER JOIN SC ON (Stu.Sno = SC.Sno)
```

获得的结果为：

Stu.Sno	Sname	Sdept	Cno	Score
201215121	张三	CS	1	92
201215122	李四	IS	NULL	NULL
...	...	...	...	...

多表连接

即进行多次两个表的连接

eg.

```
SELECT Stu.Sno, Sname, Cname, Score
FROM Stu, SC, Course
WHERE Stu.Sno = SC.Sno AND SC.Cno = Course.Cno;
```

嵌套查询

一个 `SELECT-FROM-WHERE` 为一个 **查询块**，在每个查询块的 `WHERE` 或 `HAVING` 子句的条件中可以嵌套查询块（允许多层嵌套），如：

```
SELECT Sname
FROM Stu
WHERE Sno IN(
    SELECT Sno
    FROM SC
    WHERE Cno = '1'
);
// 查询选了id为1的课的同学的姓名
```

其中，下层查询块是嵌套在上层查询块中的，称上层查询块为 **外层查询** 或 **父查询**，下层查询块为 **内层查询** 或 **子查询**。

注意：

- 子查询中SELECT的列必须与父查询中WHERE的列相同。
- 子查询不允许使用 `ORDER BY`，只能对最外层的查询排列。

带有 IN 谓词的子查询

- **不相关子查询**
  - 子查询的条件不依赖于父查询。
  - 先执行子查询，将子查询的结果作为父查询的条件。

eg.
对于：

```
SELECT Sno, Sname, Sdept
FROM Stu
WHERE Sdept IN(
    SELECT Sdept
    FROM Stu
    WHERE Sname = '张三'
); // 查询与张三同系的学生的学号、姓名和所在系
```

相当于先执行：

```
SELECT Sdept
FROM Stu
WHERE Sname = '张三';
```

结果为 `('CS')`

然后执行：

```
SELECT Sno, Sname, Sdept
FROM Stu
WHERE Sdept IN ('CS')
```

部分 嵌套查询可以转换为连接查询，如上方的例子可以转为：

```
SELECT s1.Sno, s1.Sname, s1.Sdept
FROM Stu s1, Stu s2
WHERE s1.Sdept = s2.Sdept AND s2.Sname = '张三';
```

带有比较运算符的子查询

若可以确定子查询的结果只有一行，则可以使用比较运算符进行连接。

如：

```
SELECT Sname, Sdept
FROM Stu
WHERE Sdept = (
    SELECT Sdept
    FROM Stu
    WHERE Sname = '张三'
); // 查询张三所在系的所有学生的姓名和所在系


SELECT Sno, Cno
FROM SC x
WHERE Score > (
    SELECT AVG(Score)
    FROM SC y
    WHERE x.Cno = y.Cno
); // 查询每个学生选修课程超过平均分的课程号
```

在上方这个例子中，子查询的 WHERE 子句 `x.Cno = y.Cno` 引用了父查询的列 `x.Cno`，这意味着子查询的执行结果**不是固定的**，而是**依赖于外部查询当前正在处理的行**，这样与查询相关的子查询称为 **相关子查询**，整个查询称为 **相关嵌套查询**。

处理过程：

- 遍历父查询表的行，假设遍历到某一行的Cno=1；
- 判断当前行是否满足父查询 `where score > (子查询)` 的条件
  - 进入子查询，使用当前父查询行的值进行子查询中的where条件判断
  - 即 `where y.Cno = 1` ，查询出所有Cno=1的成绩
  - 使用聚集函数 `AVG()` 计算出平均值，返回给父查询
  - 判断父查询的条件是否成立
- 如果成立，则将该行加入结果表中；否则继续父表下一行的比较

带有 ANY/SOME ALL 谓词的子查询

若子查询返回单个值可以用比较运算符，若返回多个值则需要使用 `ANY` （有些系统用 `SOME` ）或 `ALL` ：

谓词	含义
ANY	
>ANY	大于子查询结果中的某个值
<ANY	小于子查询结果中的某个值
>=ANY	大于等于子查询结果中的某个值
<=ANY	小于等于子查询结果中的某个值
=ANY	等于子查询结果中的某个值
!=ANY	不等于子查询结果中的某个值
ALL	
>ALL	大于子查询结果中的所有值
<ALL	小于子查询结果中的所有值
>=ALL	大于等于子查询结果中的所有值
<=ALL	小于等于子查询结果中的所有值
=ALL	等于子查询结果中的所有值（无意义）
!=ALL	不等于子查询结果中的所有值

eg.

```
SELECT Sname, Sage
FROM Stu
WHERE Sage < ANY (
  SELECT Sage
  FROM Stu
  WHERE Sdept='CS')
AND Sdept!='CS';
// 查询非计科系中比计科系任意一个学生年龄小的学生的姓名和年龄
// 也可以等价于小于最小值
```

带有 EXISTS 谓词的子查询

`EXISTS`，即存在量词 $\exists$，用于判断子查询的结果集中是否存在满足属于、子集、非空等条件，最终返回逻辑值 `true` 或 `false`。

由 EXISTS 引出的子查询的目标列表达式无实际意义，因此常用 `*` 表示。

`NOT EXISTS` 同理，若结果为空集则返回 `true`，否则返回 `false`。

eg.

```
SELECT Sname
FROM Stu
WHERE EXISTS (
  SELECT *
  FROM SC
  WHERE Sno = Stu.Sno AND Cno = '1'
); // 查询选修了1号课程的学生姓名
```

处理过程：

- 遍历父查询表的行，假设遍历到某一行的Sno=114514；
- 判断使用当前行进行子查询的结果是否为空
  - 进入子查询，使用当前父查询行的值进行子查询中的where条件判断
  - 即 `where Sno=114514`，查询出所有学生id为114514的成绩
  - 还有AND条件：且选修了1号课程
- 如果子查询结果非空，则返回true，把当前行的Sname加入结果表，否则返回false，跳过当前Sname

这个查询也可以等价为不相关IN查询：

```sql
SELECT Sname
FROM Stu
WHERE Sno IN (
  SELECT Sno
  FROM SC
  WHERE Cno = '1');
```

注意：

- 部分(NOT)EXISTS查询无法用其他形式的子查询等价替换
- 所有其他形式的子查询都可以用(NOT)EXISTS替换
- EXISTS子查询只关心内层查询是否有返回值，而不关心返回值的内容

SQL语言中没有全称量词 $\forall$，但是可以转换为：

$$\forall x P(x) \equiv \neg \exists x (\neg P(x))$$

即涉及到全称量词的查询可以转换为两层NOT EXISTS的子查询，如：

```sql
SELECT Sname
FROM Stu
WHERE NOT EXISTS (
  SELECT *
  FROM Course
  WHERE NOT EXISTS (
    SELECT *
    FROM SC
    WHERE Sno = Stu.Sno
      AND Cno = Course.Cno
  )
);
// 查询选了所有课的学生
```

查询过程：

1. **外层查询 (针对每个 `Stu`)**:
   - 从 `Stu` 表中选出一个学生，比如学号为 `S1`。
2. **外层 `NOT EXISTS` (检查“是否存在未选的课”)**:
   - 开始执行其内部的子查询（中间层查询），目标是看这个子查询会不会返回 *任何* 结果。
   - 如果中间层子查询 *没有返回任何行*，则外层的 `NOT EXISTS` 条件为 `TRUE`，学生 `S1` 被选中。

- 如果中间层子查询 *返回了至少一行*，则外层的 `NOT EXISTS` 条件为 `FALSE`，学生 `S1` 不被选中。

3. **中间层查询 (针对学生 `S1` 和所有 `Course`)**:
   - 从 `Course` 表中选出一门课程，比如课程号为 `C1`。
   - **内层 `NOT EXISTS` (检查“是否 S1 *没有* 选 C1”)**:
     - 开始执行其内部的子查询（最内层查询）。
     - **最内层查询 (检查 `SC` 表)**: 在 `SC` 表中查找是否存在 `Sno = S1` 且 `Cno = C1` 的记录。
       - **如果找到记录**: 说明 `S1` 选了 `C1`。最内层查询返回行，内层 `NOT EXISTS` 为 `FALSE`。
       - **如果没找到记录**: 说明 `S1` *没* 选 `C1`。最内层查询不返回行，内层 `NOT EXISTS` 为 `TRUE`。
   - **中间层 `WHERE` 条件**: 如果内层 `NOT EXISTS` 为 `TRUE` （即 `S1` 没选 `C1`），那么课程 `C1` 满足中间层查询的 `WHERE` 条件。
   - **中间层结果**: 这个查询会收集所有 `S1` *没有* 选修的课程。如果 `S1` 选了所有课，这个查询返回空集（零行）；如果 `S1` 缺了课，这个查询会返回那些缺了的课程。
4. **回到外层 `NOT EXISTS`**:
   - 现在知道了中间层查询的结果（对于学生 `S1`，他没选的课程列表）。
   - 如果这个列表是空的（中间层返回零行），说明 `S1` 没有“没选的课”，即选了所有课。外层 `NOT EXISTS` 为 `TRUE`，选中 `S1`。
   - 如果这个列表非空（中间层返回至少一行），说明 `S1` 至少缺了一门课。外层 `NOT EXISTS` 为 `FALSE`，不选 `S1`。
5. **循环**: 重复步骤 1-4，对 `Stu` 表中的下一个学生进行判断，直到所有学生都被检查完毕。

```sql
SELECT Cname
FROM Course
WHERE NOT EXISTS (
  SELECT *
  FROM Stu
  WHERE NOT EXISTS (
    SELECT *
    FROM SC
    WHERE Sno=Stu.Sno
      AND Cno=Course.Cno
  )
); // 查询所有学生都选修的课程
```

问题的逻辑转换

- **原始目标 (使用全称量词):** 找出所有课程 `c`，对于 *所有* 学生 `s`， `s` 都选修了 `c`。

    - $\{c \in Course \mid \forall s \in Stu, \text{s 选修了 c}\}$
- **第一次转换 (利用否定):** 找出所有课程 `c`，*不存在* 任何一个学生 `s`，使得 `s` *没有* 选修 `c`。
    - $\{c \in Course \mid \neg\exists s \in Stu, \neg(\text{s 选修了 c})\}$
- **`s` 选修了 `c` 的含义:** 在 `SC` 表中存在记录 `(s.Sno, c.Cno)`。
- **`s` 没有选修 `c` 的含义:** 在 `SC` 表中不存在记录 `(s.Sno, c.Cno)`。
    - $\neg(\text{s 选修了 c}) \equiv \neg\exists sc \in SC(sc.Sno = s.Sno \wedge sc.Cno = c.Cno)$

也可以用聚集函数实现：

```
SELECT Cname
FROM Course
WHERE
```