9. SQL：查询

SQL (Structured Query Language) 包含了三个子语言：

- 数据定义语言；
- 数据操作语言；
- 数据控制语言。

SQL 查询：SQL 查询是不改变数据库的，具有以下形式：

```
SELECT desired attribute list L FROM table names R, S, ... WHERE conditions C
and theta;
```

在 RA 模型中，即为 $\pi_L(\sigma_C(R \bowtie_\theta S))$。特殊情况下，单关系查询：

```
SELECT atrributes/expression-list L FROM R WHERE condition-C;
```

在 RA 模型中，即为 $\pi_L(\sigma_C(R))$。操作语义：R 看成元组变量，每次可以赋值为表中一个元组且遍历所有元组；对于表中的每个元组，检查是否满足 `WHERE` 后的式子，若满足则将 `SELECT` 后的属性输出。

- 如果 `SELECT` 后接 `*`，则表示所有属性；

  ```
  SELECT * FROM S;
  ```

- 可以在 `SELECT` 后接 `AS` （可选）来对属性名字重命名；

  ```
  SELECT sno as StuNumber FROM S;
  ```

- 可以用表达式来创建新的属性（最好在其后接 `AS` 进行重命名）；

  ```
  SELECT name, 2020 - age AS BirthYear FROM S;
  ```

- 也可以使用常量表达式来创建新的属性（可能在报表中有用）。

  ```
  SELECT name, 2020 - age AS BirthYear, 'A.D.' AS AD FROM S;
  ```

条件表达式：在 SQL 语言中，除了普通的条件表达式，也可以使用 `BETWEEN ... AND ...`，`IN`，`AND`，`OR`，`NOT` 等表达式来表达。如

```
age BETWEEN 20 AND 30
dept IN ('CS', 'EE', 'MA')
```

字符串运算：SQL 的字符串被单引号描述 `'str'`，字符串中的单引号用两个单引号 `''` 描述，如 `'I''m OK'`。

- 字符串的比较是字典序比较；
- 字符串拼接用 `||` 来操作，如 `str1 || str2`；
- **字符串匹配**：找到匹配规律 `p` 的字符串，用 `LIKE` 语句。
  - 规律 `p`：一个可包含 `%` 与 `_` 的字符串；其中，`%` 可匹配任意字符串；`_` 可匹配任意单个字符；其余字符匹配自身。
  - `s LIKE p` 为真当且仅当 `s` 能被 `p` 匹配；
  - `s NOT LIKE p` 为真当且仅当 `s` 不能被 `p` 匹配；
  - 使用 `ESCAPE` 语句来进行转义（需要用到 `%` 或 `_` 原意时），如下内容表达的是判断 `s` 是否具有 `%[任意字符串]%` 的形式。

    ```
    s LIKE '#%%#%' ESCAPE '#'
    ```

日期和时间：SQL 支持 `DATE`，`TIME`，`TIMESTAMP` 类型，他们为常量，如

```
DATE '1931-09-18'
TIME '22:20:01.23'
TIMESTAMP '1931-09-18 22:20:01'
TIME WITH TIME ZONE '12:00:00+8:00'
```

可以用 `<, >, =` 等来比较时间的大小（早晚），用 `-` 表示时间的差。

空值：特殊的值为空值 `NULL`，可以用于任何一个类型，可能在插入、外连接等情况下创建。

- 空值出现在算术表达式中，结果一定是空值；空值出现在比较表达式中，结果为 `UNKNOWN`；
- 空值可能代表一些逻辑意义上恒等表达式不成立，如 `0 * x = 0`；
- NULL 不是一个常数，因此其不能被显式用于操作数，如不能使用 `NULL + 3` 等等。
- 判断某个数 `x` 是否为 `NULL` 不能使用 `x = NULL`，需要使用 `x IS [NOT] NULL`。

三值逻辑：具有三个值的逻辑，分别是 `TRUE`、`FALSE`、`UNKNOWN`。

- 逻辑运算规则：令 `TRUE = 1`，`FALSE = 0`，`UNKNOWN = 1/2`；于是 `AND` 为 $\min$，`OR` 为 $\max$，`NOT` 为 $1-x$。
- 在 `SELECT ... FROM ... WHERE` 中，一个元组被输出当且仅当 `WHERE` 后表达式为 `TRUE`（注意空值的问题），如假设 `zhao` 的年龄为空，那么如下操作将不会出现 `zhao`：

  ```
  SELECT name FROM S WHERE age < 18 OR age >= 18;
  ```

在结果上进行排序：在 `SELECT ... FROM ... WHERE` 操作中运用 `ORDER BY` 语句进行排序，一般在 `ORDER` 中用行序号进行表示（特别用于表达式计算得到的列）。对每个属性，可用 `ASC` 和 `DESC` 分别表示升序、降序，其中 `ASC` 可省略。故通用形式如下：

```
SELECT ... FROM ... WHERE ...
ORDER BY A1[[ASC] | DESC], A2[[ASC] | DESC], ...;
```

一些典型例子如下：

```
SELECT sno, grade FROM SC
WHERE cno = 'CS145'
ORDER BY grade DESC, sno;
SELECT name, dept, 2020 - age FROM S
ORDER BY 3, 1;
SELECT name FROM S
ORDER BY 2020-age;
```

多关系查询

- **多关系查询中的重名现象**：使用关系名 + 点 + 属性名来区分不同关系的属性。
- **引用两个或多个同一关系**：可以使用 `FROM ... AS ...` 来重命名（`AS` 可省略），如

  ```
  SELECT firstS.sno, secondS.sno
  FROM S firstS, S secondS
  WHERE firstS.name = secondS.name AND firstS.sno < secondS.sno;
  ```

- 多关系查询时，实际上有三种理解方式：循环嵌套枚举、并行赋值以及 RA 模型中的笛卡尔积。因此，可能带来一些背离直觉的结果。如：

  ```
  SELECT R.A FROM R, S, T
  WHERE R.A = S.A OR R.A = T.A;
  ```

  当 `T` 为空时，结果为空，并不是 $R \cap S$。原因是：

  - 循环嵌套枚举时，循环 T 迭代了 0 次；
  - 并行赋值时，T 不能被赋值；
  - RA 模型中，笛卡尔积为空。

集合论操作：可以采用 `UNION`，`INTERSECT` 以及 `EXCEPT` 分别表示并集、交集、对称差。

子查询：一个是其他查询的一部分的 `SELECT-FROM-WHERE` 查询，子查询同样可以有自身的子查询。一个简单的子查询例子为用集合论操作连接的两个子查询。子查询可以用在另一个查询的 `WHERE` 表达式中，并且其可以返回一个原子值、单个元组或一个关系；子查询同样可以用在 `FROM` 表达式中，一般返回的是关系并且需要换名。

- 标量子查询：如果一个查询产生了单个值，则可以被看成标量常量，即如下表达式成立：`v = (SELECT ... FROM ... WHERE)`；一般用括号来包围子查询。例如：

  ```
  SELECT name
  FROM S
  WHERE age = (SELECT age FROM s WHERE sno = 'S!');
  ```

    - **作用域规则**：每个属性指代的关系的是最近的包含该属性的关系；可使用别名区分不同属性所在关系。
    - 必须用括号将子查询包围住。
- 元组子查询：如果一个子查询仅产生了一个元组，则可以被看成单个元组，即如下表达式成立：`(v1, v2) = (SELECT ... FROM ... WHERE)`。
- 关系子查询：一般来说，子查询产生一个关系 `R`，即可以使用如下表达式：
    - `t IN R`，`t NOT IN R`，`NOT (t IN R)`：判断 `t` 是否在 `R` 中。
    - `EXISTS(R)`，`NOT EXISTS(R)`：判断 `R` 是否为空。
    - `t theta ALL R`，`t theta ANY R`（`t theta SOME R`）：量词 $\forall, \exists$，满足条件 `theta`。
        - 如果 $R = \varnothing$，则 `t theta ALL R` 为真，`t theta SOME R` 为假。

包含元组的查询：可以使用括号+逗号表示一个标量元组，如 `(123, 'foo') IN R`。

- 注意：成员属性必须匹配！

连接表达式：SQL 中提供了许多连接运算子来构造连接，如

```
R CROSS JOIN S
R JOIN S ON theta
R NATURAL JOIN S
R OUTER JOIN S
```

这些 `JOIN` 表达式可以被用在任何需要连接而成的关系的地方，同时本身也可以作为查询。

- 外连接可以在 `OUTER JOIN` 前加入可选的 `NATURAL` 选项表示自然外连接；外连接可以在 `OUTER JOIN` 后加入外连接的条件；外连接可以在 `OUTER JOIN` 前加入 `LEFT`、`RIGHT` 或 `FULL` 表示左外连接、右外连接以及完全外连接（默认为完全外连接）。
- 可以在任意 `JOIN` 后加入 `ON` 表示条件。

Bag 语义：SQL 关系事实上是 Bag，所以 SQL 的询问 (query) 事实上不会去除重复元素，我们可以在 `SELECT` 后加入 `DISTINCT` 来强制对询问结果进行去重，例如：

```
SELECT DISTINCT sno FROM SC;
```

但是在集合操作 `UNION`，`INTERSECT` 与 `EXCEPT` 中，默认是去重的，可以在集合操作后加入 `ALL` 来强制不对结果进行去重，例如

```
R UNION ALL S
R INTERSECT ALL S
R EXCEPT ALL S
```

聚合函数：可以对列（属性）进行聚合操作，得到一个单一的值。其中，聚合函数可以是 `SUM()`，`AVG()`，`MIN()`，`MAX()`，`COUNT(A)`，`COUNT(*)`；其中聚合函数的输入是一个属性或标量表达式，聚合函数经常用在 `SELECT` 表达式的查询中。例如：

```sql
SELECT SUM(grade) FROM SC;
SELECT AVG(age) FROM S WHERE dept = 'CS';
SELECT COUNT(*) FROM S;
SELECT COUNT(age) FROM S;
SELECT MAX(grade) FROM SC, C WHERE SC.cno = C.cno AND C.name = 'DB';
```

可以使用 `DISTINCT` 在聚合函数中去重，可以与任何聚合函数搭配，但是一般与 `COUNT()` 连用，因为这样才有实际意义（注意这里，聚合函数 `COUNT` 默认是不去重的）。

```sql
SELECT COUNT(DISTINCT age) FROM S WHERE dept = 'CS'
```

分组：将元组根据某些属性分组，其中相同的为一组，经常与聚合函数连用。用法如下：

```sql
SELECT ... FROM ... WHERE ... GROUP BY list-of-grouping-arguments;
```

- 操作语义：对于如下操作

  ```sql
  SELECT aggregation
  FROM R, S, ...
  WHERE C
  GROUP BY A;
  ```

  语义为：

  1. 计算 $\sigma_C(R \times S \times \cdots)$；
  2. 根据 A 进行分组；
  3. 计算每组的聚合值；
  4. 对每组的结果，得到一个结果关系的元组。

在分组聚合中使用 `SELECT`：如果使用了聚合函数，那么 `SELECT` 所选择的任何元素必须要么出现在聚合函数中，要么出现在分组属性中（即 `GROUP BY` 后）；否则将会出现一个元组的属性为多个值的情况，不符合 SQL 规则。例如，以下的语句是不合法的

```sql
SELECT name, MIN(age) FROM S GROUP BY dept;
SELECT name, AVG(grade) FROM S, SC WHERE S.sno = SC.sno GROUP BY sno;
```

而如下语句是合法的：

```sql
SELECT name, AVG(grade) FROM S, SC WHERE S.sno = SC.sno GROUP BY name, sno;
```

聚合函数中的 `NULL`：在聚合函数中，`NULL` 被忽略，即其不对任何聚合函数产生贡献；但是在进行分组中，NULL 被当作一个正常值。如果一列中全部均为 `NULL` 的元素，则除 `count` 外的聚合结果均为 `NULL`，`count` 的聚合结果为 `0`。

`HAVING` 语句：我们可以使用 `GROUP BY ... HAVING condition` 来对于分组进行选择，其中条件 `condition` 中一般含有聚合函数，这个聚合函数仅对分组内的元组进行操作。`HAVING` 与 `WHERE` 的区别如下：

- `WHERE` 是对元组的选择，在分组操作之前执行；
- `HAVING` 是对分组的选择，在分组操作之后进行。

一些例子如下：

```
SELECT sno, AVG(grade) FROM SC
  GROUP BY sno HAVING MIN(grade) >= 60;
SELECT dept, AVG(age) FROM S
  WHERE age > 18 GROUP BY dept
  HAVING COUNT(*) >= 50;
SELECT cno, AVG(grade) FROM SC
  GROUP BY cno HAVING COUNT(*) >= 50 OR
    cno IN (SELECT cno FROM C WHERE name = 'DB');
```

询问的总体格式

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ...
```

一个例子：

```
SELECT name, '''s avg=', AVG(grade) FROM S, SC
  WHERE S.sno = SC.sno AND grade >= 60
  GROUP BY S.sno, name
  HAVING SUM(grade) >= 600
  ORDER BY 3 DESC, name;
```

即将每个 除不及格外总分大于 600 的学生的学号与平均分输出，并按照平均分降序排序。

SQL 中 ÷ 的实现：例如，找到上了所有课的所有学生（利用双重否定，不存在课程中不存在这个学生）。

```
SELECT DISTINCT sno
  FROM SC X WHERE NOT EXISTS
    (SELECT * FROM C WHERE NOT EXISTS
      (SELECT * FROM SC Y WHERE Y.cno = C.cno AND Y.sno = X.sno));
```

10. SQL：修改

插入一个元组：我们可以使用 `INSERT INTO R(A1, A2, ..., An) VALUES (v1, v2, ..., vn)` 来插入一个元组。

- 可以仅给部分列提供值，需要在 `A1, A2, ..., An` 注明提供值的列，其中 `vi` 为给 `Ai` 列提供的数据。如果对全部列均提供值，则可以省略 `(A1, A2, ..., An)`。
- 没有被提供值的属性为默认值 (default)，如果没有对属性默认值进行修改则为 `NULL`。
- `(A1, A2, ..., An)` 可以不按照关系的顺序，只要满足 `vi` 与 `Ai` 对应即可。

一些例子如下：

```
INSERT INTO Student VALUES ('008', 'Chow', 18, 'CS');
INSERT INTO Student(sno, name) VALUES ('008', 'Chow');
INSERT INTO Student(name, sno, dept) VALUES ('Chow', '008', 'CS');
```

插入许多元组：我们可以使用 `INSERT INTO R(A1, A2, ..., An) SELECT ... FROM ... WHERE` 来插入许多元组，具体用法见下：

- 可以仅给部分列提供值，对全部列提供值可以省略标注，数据与属性按顺序一一对应（与插入一个元组相同）。
- SQL 语义中，在操作的最后才会插入所有即将插入的元组（新插入的不会影响 `SELECT` 的结果）。

一些例子如下：

```
CREATE TABLE CSNameList (name char(10));
INSERT INTO CSNameList SELECT name FROM Student WHERE dept = 'CS';
```

删除元组：我们可以使用 `DELETE FROM R WHERE condition` 来删除 `R` 中满足条件 `condition` 的元组。

- 如果 `WHERE condition` 省略，那么删去所有元组，但是仍然存在一张空表！
- SQL 语义中，在操作的最后才会删除所有即将删除的元组（已删除的不会影响 `condition` 的结果）。
- 在 SQL 中，删除元组时不会真正删除并将下面的元组上移，会打标记删除，在插入时重用并在特定时期对表重构。

一些例子如下：

```
DELETE FROM Student WHERE sno = '008';
DELETE FROM Student WHERE age > 30;
DELETE FROM SC;
DELETE FROM Student S WHERE EXISTS
  (SELECT * FROM SC WHERE SC.sno = S.sno AND grade = 0);
DELETE FROM Course X WHERE EXISTS
  (SELECT * FROM Course Y WHERE X.cno <> Y.cno AND X.name = Y.name);
```

其中，在第 5 条中，作用是“删除所有具有相同课名单但课号不同的课”；如果理解为循环语义（边做边删），那么会保留其中一个课程，显然是与 SQL 语义不同的。

更新：我们可以使用 `UPDATE R SET list-of-assignments WHERE condition` 来通过 `assignments` 更新 `R` 中满足条件 `condition` 的元组。

- 一个 `assignment` 可以写为：`attribute = expression`；
- SQL 语义中，在操作的最后才会更新所有即将操作的元组（已操作的不会影响 `condition` 的结果）。

一些例子如下：

```
UPDATE S SET dept='CS', age=age+1 WHERE sno='007';
UPDATE S SET dept='CompSci' WHERE dept='CS';
```

10.5 SQL：事务管理

数据库操作中的复杂性

- 可能有许多用户同时操作同一个数据库；
- 可能存在出现错误而未完成的操作。

数据库经常被许多用户同时访问（修改/询问）；如果不加以控制，那么一些交互式事件会导致一些不希望产生的结果，如更新丢失、读到脏数据等等。本地正确的操作因为交叉执行，可能在全局上并不是正确的。

串行执行：对每个用户的指令顺序执行，在执行中不进行其他用户指令的处理。串行执行中，不存在任何上述问题，但是效率极低。

并发执行：对用户的指令并发执行，在执行一个用户指令后可以插入其他用户的指令。并发指令效率高，但是可能导致一些问题。

可串行化的 (serializable)：并发执行，但是结果看上去与串行相同（不出现问题），如加锁进行并发控制等。并发的且可串行化的交互执行是我们所需要的执行方式。

故障：即使只有单用户在操作数据库，数据库仍然可能出现故障（硬件或软件问题），导致我们不希望的结果；如银行转账中，在对两个用户数据进行修改间出现故障，则会出现数据异常。

事务：一个或多个数据库操作组成了一个操作的逻辑单位，这些操作需要被原子性执行。

ACID 事务：ACID 事务满足

- 原子性 (Atomic)：要么执行完成事务中的所有操作，要么没有进行任何操作；
- 一致性 (Consistency)：数据库的限制仍然满足；
- 独立性 (Isolated)：对用户来说看上去像每次执行一个指令；
- 持久性 (Durable)：操作的影响在故障后仍然存在（数据库一般存在内存上，并立即不会存到磁盘中，故障时可能会丢失信息，持久性要求出现故障后，数据仍然可以恢复）。

事务

- SQL 允许将若干语句组合为一个事务，可显式执行：`START TRANSACTION mode [{, mode}...]`。
- 事务可以显式结束，如显式的 `COMMIT` 提交修改，或显式 `ROLLBACK` 回滚到某个前期状态；或隐式 `ROLLBACK`，即故障后系统自动产生的 `ROLLBACK`；
- 事务的模式：
  - 访问层面：`READ ONLY` / `READ WRITE`；
  - 隔离层面：`READ UNCOMMITTED` / `READ COMMITTED` / `REPEATABLE READ` / `SERIALIZABLE`；
  - 通过 `SET TRANSACTION mode [{, mode}...];` 来设定模式。
- 事务隔离级别：SQL 中允许对事务并发性的灵活控制，可串行化条件并非总是被满足。一个事务 T 的隔离级别仅仅影响到 T 中对于数据的视角，并不会影响到其他事物对于数据的视角。

脏读取

- 脏数据：被某个事务修改了，但是该事务还没有提交；
- 脏读取：T_2 读取了 T_1 修改过但未提交的脏数据；
  - 风险：T_1 可能在 T_2 提交后回滚，则脏数据不应该存在。

隔离级别 1: RU (Read Uncommited)：SQL 通过如下指令设置隔离级别为 Read Uncommitted。

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

- 这种隔离级别允许“脏读取”；默认的访问级别：Read Only；
- 如果我们该事务需要 R/W 权限，而同时使用 `READ UNCOMMITED` 则需要格外小心可能出现的情况。该指令可以与前面的指令合并为：

```
SET TRANSACTION READ WRITE ISOLATION LEVEL READ UNCOMMITTED;
```

隔离级别 2: RC (Read Commited)：SQL 通过如下指令设置隔离级别为 Read Comitted。

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

- 不允许“脏读取”，不过会造成“不可重复读取” (non-repeatable read)。
  - 不可重复读取：两次对于相同询问的执行结果可能在某个元组上不同。
- 默认的访问级别：R/W。

隔离级别 3: RR (Repeatable Read)：SQL 通过如下指令设置隔离级别为 Repeatable Read。

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

- 不允许“脏读取”与“不可重复读取”，不过会造成“幽灵现象” (phantom)。
  - 幽灵现象：两次对于相同询问的执行结果可能增加/删除了部分元组。
  - 与“不可重复读取”的区别：“不可重复读取”一般是由于修改造成，而“幽灵现象”一般是由于增加/删除造成。
- 默认的访问级别：R/W。

隔离级别 4: Serializable：SQL 通过如下指令设置隔离级别为 Serializable。

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

- SQL 默认事物隔离级别；不允许“脏读取”与“不可重复读取”，不会造成“幽灵现象”。
- 默认的访问级别：R/W。

注意：当然，隔离级别 1 也会出现“不可重复读取”现象；隔离级别 1 与 2 也会出现“幽灵”现象。