11. 约束

完整性约束 通过限制数据库中的某些值来定义数据库中的合法状态。

- 对允许的数据的值的约束；
- 在表的定义时候加入完整性约束。

为什么需要完整性约束

- 发现表项的错误；
- 作为数据库更新时的正确性评判准则；
- 保证数据库中数据的一致性；
- 告诉系统关于数据的信息，以便与系统处理（如创建索引等）。

如何执行完整性约束？IC 是一个储存在数据库中表达式，且 DBMS 会在合适的时间执行这个约束；如果 IC 违反则拒绝该操作。

不同种类的完整性约束

- 表约束：键值约束、引用完整性约束、表检查约束；
- 定义域约束；
- 断言。

键值约束：表中不能存在相同的键值。

- 有两种定义键值的方式：使用 `PRIMARY KEY` 与使用 `UNIQUE`。一个关系至多含有一个 `PRIMARY KEY`，不过可以有若干个 `UNIQUE` 的属性；
- `PRIMARY KEY` 属性不可为 `NULL`；而 `UNIQUE` 属性可以是 `NULL`；并且两个 `NULL` 被视为不同的。
- **执行键值约束**：仅当插入或修改后进行检查。
    - 一个基于键值属性的**索引**对于高效检查来说至关重要，否则需要对整张表做一遍扫描；DMBS 通常自动为键值属性创建索引。

引用完整性约束 (RI)：数据必须在数据库内保持合理，不只是在单个表中。

- 例如：`SC` 表中的 `('007', 'CS123', 88)` 必须在 `Student` 表中存在 `'007'` 且 `Course` 表中存在 `CS123` 时才合理。
- **外键 (Foreign Key, FK)**：在关系 R 中，如果其属性 A 引用了另一个属性 S 的键值，那么 A 是 R 的外键。
- **引用完整性约束**：任何出现在关系 R 的外键中的非空值必须出现在原有关系 S 的键值中。
- 定义引用的方式：
    - 属性层面：

```
CREATE TABLE SC (
  sno char(10) REFERENCES S(sno),
  cno char(10) REFERENCES C(cno),
  grade integer
);
```

- 列表层面

```
CREATE TABLE SC (
  sno char(10),
  cno char(10),
  grade interger,
  FOREIGN KEY (sno) REFERENCES S(sno),
  FOREIGN KEY (cno) REFERENCES C(cno)
);
```

- 违反引用完整性约束的情况：
  1. 将不存在的 FK 值插入 `R(FK)` 中；
  2. 用不存在的 FK 值更新 `R(FK)`；
  3. 删除 `S(K)` 中的 `K` 值；
  4. 更新 `S(K)` 中的 `K` 值。
- **执行引用完整性约束**：4 种策略
  - 默认策略（处理 1、2、3、4 情况）：直接拒绝违反 RI 的操作；
  - Cascade 策略（处理 3、4 情况）：在删除/更新 `K` 的同时删除/更新 `FK`；
  - Set-Null 策略（处理 3、4 情况）：在删除/更新 `K` 的同时将对应的 `FK` 置 `NULL`；
  - Set-Default 策略（处理 3、4 情况）：在删除/更新 `K` 的同时将对应的 `FK` 置默认值（需要满足 `FK` 的默认值已经在 `K` 中）。

  在引用完整性定义后使用 `ON [option] [policy]` 来规定采取的策略，可以采取多种策略，如：

```
CREATE TABLE SC (
  sno char(10) REFERENCES S(sno)
               ON DELETE SET NULL
               ON UPDATE CASCADE,
  ... ...
);
```

  - 一般来说，`CASCADE` 适用于更新；`SET NULL` 适用于删除。
- **自定义检查时间**：检查的时间可以自定义（进行延迟），一般有这几种选项：
  - `DEFERRABLE`：检查时间将会被推迟，其后设定检查时间的初始值：
    - `INITIALLY DEFERRED`：在事务的最后（事务提交之前）检查引用完整性；
    - `INITIALLY IMMEDIATE`：在每次修改后立即检查引用完整性。

    初始值在之后使用 `SET CONSTRAINT constraint_name DEFERRED | IMMEDIATE` 修改。

- `NOT DEFERRABLE`（默认）：在每次修改后立即检查引用完整性。（注意其与 `INITIALLY IMMEDIATE` 的差别是这里不能修改检查时间，而前者可以修改检查时间至 `DEFERRED`。）

自定义检查时间的原因：循环限制。

```
CREATE TABLE R(a integer UNIQUE);
CREATE TABLE S(a integer PRIMARY KEY REFERENCES R(a));
ALTER TABLE R ADD CONSTRAINT mycon FOREIGN KEY (a) REFERENCES S(a)
DEFERRABLE INITIALLY DEFERRED;
BEGIN TRANSACTION;
INSERT INTO R VALUES(1);
INSERT INTO S VALUES(1);
COMMIT;
```

非空约束：要求属性拒绝 `NULL` 值，防止出现不提供具体值的插入以及更新为空值。

- 不过可以使用 `UPDATE R SET A=NULL` 将整列置空。
- **执行非空约束**：在每次插入/更新后进行检查。需要在对应的属性后增加 `NOT NULL` 进行定义。

```
CREATE TABLE S (
  sno char(10) PRIMARY KEY,
  name char(20) NOT NULL,
  age integer,
  dept varchar(30) NOT NULL
);
```

在某些时候，`NOT NULL` 为默认选项；如果某列需要空值，则需要在其后加 `NULL` 加以说明。

基于属性的检查：可以基于属性值进行一定的检查。

- 在 `CREATE TABLE` 中需要检查的属性后加入 `CHECK (condition)` 检查其是否满足 `condition` 条件。
- 注意 `condition` 条件可以直接使用其修饰的属性，但其他属性必须使用条件中子查询得到的结果的属性。
  - 一些 DBMS 可能不支持 `CHECK` 中嵌套子查询。

```
CREATE TABLE Stud_notMH (
  sno char(10) PRIMARY KEY,
  name varchar(20),
  age integer CHECK (age > 0),
  dept char(20)
    CHECK (dept NOT IN (SELECT name from Depts WHERE location='MinHang'))
);
```

- **执行基于属性的检查**：每当任何元组的相关属性得到新的值（无论是通过 `INSERT` 还是 `UPDATE`），均会进行检查（检查更新后的属性）；拒绝一切不符合限制 `condition` 的操作。
  - 需要注意的是，条件可能由于相关表进行了修改而从真变为了假，而 DBMS 只会因为相关属

性的修改而检查，并不会因为相关表的修改而检查。一个典型的例子就是上述描述中，将 `Depts` 中的某一元组的 location 从 `'XuHui'` 更改为 `'MinHang'` 并不会使 DBMS 执行基于属性的检查。

基于元组的检查：对于关系中元组的一些限制，是表层面的检查，包括一个元组的许多属性。

- 在所有属性定义完成后加入 `CHECK condition` 声明，其中 `condition` 可使用任何关系中的属性或使用子查询得到的结果的属性。

```
CREATE TABLE Stud_CME (
  sno char(10) PRIMARY KEY,
  name varchar(20),
  age integer,
  dept char(20),
  CHECK (dept in ('CS', 'MA', 'EE') OR age < 16)
);
```

- **执行基于元组的检查**：每当插入/修改任何元组时进行检查（检查更新后的属性）；拒绝一切不符合限制 `condition` 的操作。
  - 这类检查对其他关系来说不可见。

对于限制的修改

- 命名限制：

```
CONSTRAINT name contraint-definition;
```

例如

```
sno char(10) CONSTRAINT pksno PRIMARY KEY
CONSTRAINT chktuple CHECK ...
```

- 删除/加入限制：

```
ALTER TABLE tablename DROP CONSTRAINT cname;
ALTER TABLE tablename ADD CONSTRAINT cname cons-def;
```

断言 (Assertions)：一个必须始终为真的布尔值的 SQL 表达式。

- 创建/删除断言

```
CREATE ASSERTION assertion_name CHECK (condition);
DROP ASSERTION assertion_name;
```

- `condition` 从创建断言开始至程序结束（或断言被删除）必须始终保持为真；
- 每当断言中包含的关系进行了修改，则检查断言。
  - 效率低下，因此目前大部分 DBMS 都还没有内置断言。

- 断言与基于元组的限制的比较：

```
CREATE ASSERTION avgpass
CHECK (60 <= ALL (SELECT AVG(grade) FROM SC GROUP BY sno));
```

  - 断言是全局角度的，可以用来解决关系限制中对其他关系不可见的问题；
  - 同时断言还会检查删除情况，而基于元组的限制只检查插入、更新情况。

触发器 (Trigger)

- **ECA 规则**：event, condition, action. Event 即对于数据库的修改；condition 即判断是否需要触发 trigger；action 即如果触发 trigger 需要执行的命令。
- 创建/删除触发器：

```
CREATE TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF} events
  REFERENCING {OLD ROW | OLD TABLE} AS oldvar
              {NEW ROW | NEW TABLE} AS newvar
  FOR EACH {ROW | STATEMENT}
  WHEN(condition)
    actions;
DROP TRIGGER trigger_name;
```

- **Event 事件**：包括 `INSERT ON R`, `DELETE ON R`, `UPDATE ON R`, `UPDATE OF A1, A2, ... ON R` 等等；定义的 events 会触发 trigger。
- **Condition 条件**：当条件为真时，立即执行 `action`；在之前介绍的 `CHECK` 以及后续操作之前，执行完 trigger 的指令。
  - 激活触发器并不意味着一定执行 `action` —— 需要 `condition` 为真！
  - 如果是无条件触发，则 `WHEN(condition)` 可省略。
- **Actions 动作**：动作是一条或多条 SQL 命令，如果多于一条 SQL 命令，使用 `BEGIN ... END` 表示起始位置和终止位置，如：

```
BEGIN
  stmt1;
  stmt2;
  ...
  stmtN;
END
```

- **元组级别与指令级别**
  - `FOR EACH ROW` 为元组级别：对每个修改的元组，执行一次触发器；
  - `FOR EACH STATEMENT` 为指令（事务）级别（默认）：对每条修改了元组的指令，在修改后仅执行一次触发器。
- **`REFERENCING` 语句**：`condition` 与 `action` 中可以使用被修改前的旧值与修改后的新值，可以使用 `REFERENCING ... AS ...` 来起别名使引用方便。在使用元组级别时，常使用 `ROW`；在使用指令级别时，由于不能直接访问元组，只能使用 `TABLE`。例如：

    - 删除元组时：`OLD ROW (TABLE) AS ot`；
    - 插入元组时：`NEW ROW (TABLE) AS nt`；
    - 修改元组时：`OLD ROW (TABLE) AS ot NEW ROW (TABLE) AS nt`。
- 触发器的例子：

  每当插入成绩时候，选择最好的学生（满分）。

```
CREATE TRIGGER best_student
AFTER INSERT ON SC
REFERENCING NEW ROW AS newSC
FOR EACH ROW
WHEN (newSC.grade = 100)
  SELECT name, dept
  FROM S
  WHERE S.sno = newSC.sno;
```

  使用触发器作为另一种设置默认值的方式。

```
CREATE TRIGGER fixName
BEFORE INSERT ON S
REFERENCING NEW ROW AS newS
            NEW TABLE AS newSS
FOR EACH ROW
WHEN newS.name IS NULL
  UPDATE newSS SET name = 'Anonymous';
```

  使用触发器作为另一种检查引用完整性的方式。

```
CREATE TRIGGER casc
AFTER DELETE ON S
REFERENCING OLD ROW AS oldS
FOR EACH ROW
WHEN (0 < (SELECT COUNT(*) FROM SC WHERE SC.sno=oldS.sno))
  DELETE FROM SC WHERE SC.sno = oldS.sno;
```