Lecture 4: 高级SQL

Database

Author: Forliage

Email: masterforliage@gmail.com

Date: June 10, 2025

College: 计算机科学与技术学院

Abstract

本讲笔记深入探讨了高级 SQL 的关键技术与实现方法，主要内容包括：
1.SQL 数据类型与模式，介绍了用户定义类型（DOMAIN、UDT）和大对象类型（BLOB、CLOB）的创建与使用；
2.完整性约束，详细阐述了域约束、参照完整性（外键的插入、删除、更新策略及级联操作）、断言（ASSERTION）的定义与性能影响，以及触发器（TRIGGER）的设计模式与使用场景；
3.授权管理，说明了基于用户、角色和视图的细粒度权限控制机制及 GRANT/REVOKE 的使用；
4.嵌入式 SQL，剖析了在宿主语言中嵌入 SQL 的预处理流程、变量声明、单行查询、游标操作与更新语句等编程模型；
5.ODBC 与 JDBC，比较了两种跨数据库访问 API 的架构、编程接口、资源管理与异常处理方法。
通过本讲学习，读者能够掌握高级 SQL 在企业级应用中保障数据一致性、安全性与跨语言访问的核心技术，为实际开发与性能优化提供坚实基础。
（该Abstract由ChatGPT-o4-mini-high生成）

Contents

1 SQL数据类型和模式

用户定义类型：结构化数据类型

```
Create type person_name as varchar(20)

create table student
        (sno char(10) primary key,
        sname person_name,
        ssex char(1),
        birthday date)
drop type person_name
```

创建新域

```
create domain dollars as numeric(12,2) not null;
create domain pounds as numeric(12,2);
create table employee
         (eno char(10) primary key,
          ename varchar(15),
          job varchar(10),
          salary dollars,
          comm pounds);
```

大对象（例如照片、视频、CAD文件等）类型:

- blob:二进制大对象–对象是大量未解释的二进制数据集合（其解释留给数据库系统之外的应用程序）
- clob:字符大对象–对象是大量字符数据集合
- 当查询返回大对象时，返回的是指针而非大对象本身

```
create table students
      (sid char(10) primary key,
       name varchar(10),
       gender char(1),
       photo blob(20MB),
       cv clob(10KB))
```

2 完整性约束

完整性约束可防止数据库意外受损，确保对数据库的授权更改不会导致数据一致性丢失。

实体完整性、参照完整性和用户定义的完整性约束

完整性约束是数据库实例(Instance)必须遵循的

单关系上的约束：非空、主键、唯一、检查(P)

2.1 域约束

```
create domain hourly-wage numeric(5, 2)
constraint value-test check(value >= 4.00)
```

子句constraint value-test是可选的；用于指示更新违反了哪个约束。

2.2 引用完整性

约束要求：对每一个参照关系r_2中的元组t_2其外键值$t_2[\alpha]$要么为NULL，要么必须在被参照关系r_1的主键集中能够找到对应的值。换句话说，外键列的取值必须是被参照表主键列的一个子集。

形式化定义

设关系$r_1(R_1)$的主键是属性集合K_1，关系$r_2(R_2)$的一个属性子集$\alpha \subset R_2$被声明为外键，引用r_1的主键K_1。则引用完整性约束可以形式化地写成: $\Pi_\alpha(r_2) \subset \Pi_{K_1}(r_1)$

为了保证这一约束在数据修改(INSERT/DELETE/UPDATE)时始终成立，数据库系统必须在对参照表或被参照表进行插入，删除或更新曹祖前后分别做如下检查：

1.INSERT:向参照关系r_2插入新元组t_2时，若其外键列α上的值不为NULL，则必须检查该值存在于被参照关系r_1中：$t_2[\alpha] \in \Pi_{K_1}(r_1)$ 如果检查失败，插入将被拒绝。

2.DELETE：从被参照关系r_1删除一个元组t_1时，必须先查看参照关系r_2中是否存在引用t_1的元组：$\sigma_{\alpha=t_1[K_1]}(r_2)$。若此查询结果非空，则有两种策略：禁止删除：直接报错拒绝删除；级联删除；同时删除所有引用该元组的r_2中的元组。

3.UPDATE：

- 更新参照关系r_2的外键列：当修改$r_2.\alpha$中的值时，新值必须依然在被参照表$r_1.K_1$中存在，否则更新被拒绝或按”级联更新“策略将被参照行同步修改。
- 更新被参照关系r_1的主键列：当修改$r_1.K_1$中的键值时，若不采取特别策略，则会违反外键约束，常见做法是：

- 禁止更新：拒绝修改。
- 级联更新：同时将所有参照关系r_2中对应的外键值值一并更新。
- 设为NULL:将所有引用的外键列设为NULL。
- 设为默认值：将外键列设为预定义的默认值。

在SQL DDL中，可以在CREATE TABLE或ALTER TABLE语句中声明外键及其约束行为：

```
CREATE TABLE department (
    dept_id INT PRIMARY KEY,
    d_name VARCHAR(50)
);

CREATE TABLE employee(
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_ref INT,
    -- declare dept_ref as foreign key, refer department(dept_id)
    FOREIGN KEY (dept_ref)
        REFERENCES department (dept_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT
);
```

2.3 级联

```
Create table account (
    ...
    foreign key (branch-name) references branch
       [on delete cascade]
       [on update cascade]
       ...
);
```

由于存在级联删除子句，如果删除branch中的一个元组导致引用完整性约束被违反，那么该删除操作会级联到account关系，删除account中引用已删除branch的元组。

如果多个关系之间存在外键依赖链，且每个依赖都指定了级联删除，那么链一端的删除或更新操作可以在整个链中传播。

但是，如果级联更新或删除导致了无法通过进一步级联操作处理的约束冲突，系统将中

止该事务。因此，该事务及其级联操作所导致的所有更改都将被撤销。

外键属性中的NULL会使SQL引用完整性语义变得复杂，最好使用非空约束来避免。

2.4 断言

定义：断言是一条全局性的谓词，表达了数据库在任何时刻、任何更新之后都必须满足的条件，通常用于对多张关系（表）之间的复杂的约束检查。

CREATE ASSERTION <断言名> CHECK (<布尔表达式>);

检查策略：一旦咋数据库中定义了某个断言，系统就会在每次可能导致该断言失效的INSERT/DELE作之后，对断言的CHECK条件做一次完整性检查——如果断言的谓词为真，操作继续；如果断言为假，操作被拒绝并报错。

性能影响：因为每个相关更新都要重新执行一次断言检查，可能涉及对多张表或聚合计算的大量扫描，带来显著的性能开销，所以在生产系统中要慎用。

SQL本身并不提供类似”FOR ALL x,P(x)“地方直接语法，但可以利用逻辑等价式：$\forall x P(x) \equiv \neg\exists x \neg P(x)$ 在SQL中就经常写成：

```
CHECK (NOT EXISTS(
    SELECT * FROM ...
    WHERE NOT (P(x))
))
```

2.5 触发器

Trigger是一种由系统自动执行，用以在数据库发生增删改时被动触发的存储过程。它的主要作用是： 1.在数据修改时自动执行特定操作，完成复杂检查、维护派生数据、记录审计日志、或与外部系统交互等 2.实现比CHECK约束或断言更灵活的逻辑，可以跨表、多步、带条件地响应数据变化

要设计一个Trigger，必须指定两个核心要素：

1. 触发条件：
    - 事件类型：INSERT/DELETE/UPDATE
    - 时机：BEFORE/AFTER
    - 对于UPDATE，还可以细化为“仅当某个属性被修改时才触发”(AFTER UPDATE OF balance ON account)
2. 触发动作
    - 一段或多段SQL语句，定义当触发条件满足时要自动执行的操作。

- 可以引用更新前或更新后的行值

例子：银行透支处理：假设我们希望不允许账户余额为负，但允许客户“透支”：当account.balance更新为负数时，系统自动:

1. 将该账户余额设为0
2. 自动创建一笔与透支金额相等的新贷款(loan)，贷款号与账户号相同
3. 并在borrower表中登记该客户的贷款记录

SQL:1999标准语法

```
CREATE TRIGGER overdraft_trigger
  AFTER UPDATE ON account
  REFERENCING NEW ROW AS nrow
              OLD ROW AS orow
  FOR EACH ROW
  WHEN (nrow.balance < 0)
BEGIN ATOMIC
  INSERT INTO borrower (customer_name, loan_number)
    SELECT D.customer_name, nrow.account_number
      FROM depositor D
    WHERE D.account_number = nrow.account_number;

  INSERT INTO loan (loan_number, branch_name, amount)
    VALUES (nrow.account_number,
            nrow.branch_name,
            - nrow.balance);

  UPDATE account
    SET balance = 0
  WHERE account_number = nrow.account_number;
END;
```

触发器的两种粒度：

1. 行级触发器
   - 使用FOR EACH ROW或默认行为
   - 每次修改一行，就独立触发一次动作
   - 适合需要对每行做细粒度处理的场景，但如果一次性更新大量行，可能性能开销大
2. 语句级触发器

- 使用FOR EACH STATEMENT
- 对某一条DML语句（即使影响多行）只触发一次
- SQL Server中可用INSERT/UPDATE临时过渡表，或Oracle中的REFERENCING NEW TABLE AS newtab来访问所有受影响行。
- 适合统计汇总、批量维护场景，更高效。

有时，我们需要在数据库修改后执行外部世界动作，例如：

- 当库存低于阈值时，自动向供应商下采购单；
- 触发报警系统等

由于数据库的trigger不能直接调用外部系统(出于transaction一致性考虑)，常用做法是：

- 在trigger中写入"待办动作"表
- 外部守护进程定期扫描该表，执行业务层面的外部动作，并在完成后删除已处理的记录。

例子：补货trigger

```
CREATE TRIGGER recorder_trigger
  AFTER UPDATE OF level ON inventory
  REFERENCING OLD ROW AS orow NEW ROW AS nrow
  FOR EACH ROW
  WHEN (nrow.level <= (
        SELECT level
        FROM minlevel
        WHERE minlevel.item = nrow.item)
     AND orow.level > (
        SELECT level
        FROM minlevel
        WHERE minlevel.item = orow.item
     )
  )
BEGIN
  INSERT INTO orders(item, quatity)
     SELECT item, amount
       FROM recorder
       WHERE recorder.item = orow.item;
END;
```

何时不宜使用Trigger:

- 维护汇总数据、数据库复制等场景，现代数据库提供了物化视图、内置复值机制，比trigger方式更高效
- 大量触发逻辑会导致事务变长、锁等待与性能瓶颈，因此应尽量将复杂业务逻辑放在应用层或专门的批处理/实时流处理系统中。

3 授权

安全性——防止恶意窃取或修改数据的行为。

- 数据库系统级别：身份验证和授权机制仅允许特定用户访问
- 操作系统级别：操作系统超级用户可以对数据库为所欲为！需要良好的操作系统级别的安全性
- 网络级别：必须使用加密来防止：窃听(未经授权读取消息)；维装(冒充授权用户或发送据称是...的消息来自授权用户)
- 物理层面：入侵者对计算机的物理访问可能会破坏数据
- 人员层面

数据库部分内容的授权形式：

- 读取授权：允许读取数据，但不允许修改数据
- 插入授权：允许插入新数据，但不允许修改现有数据
- 更新授权：允许修改数据，但不允许删除数据
- 删除授权：允许删除数据

修改数据库模式的授权形式：

- 索引授权：允许创建和删除索引
- 资源授权：允许创建新的关系
- 修改授权：允许在关系中添加或修改属性
- 删除授权：允许删除关系

授权与视图：

- 可以为用户授予视图的访问权限，而无需授予视图定义中所有使用关系的任何访问权限
- 视图隐藏数据的能力既有助于简化系统的使用，又能通过仅允许用户访问其工作所需的数据来增强安全性。
- 可以结合使用关系级别的安全性和视图级别的安全性，以精确限制用户对其所需数据得访问

视图授权：由于为创建实际关系，创建视图不需要资源授权。视图创建者仅获得那些不会超出其已有权限的额外授权的特权。

授权从一个用户传递到另一个用户可以用授权图来表示。此图的节点的用户，图的根节点是数据库管理员。

要求：授权图中的所有边必须是数据库管理员的某条路径的一部分。

SQL中授予语句用于授予权限：

```
GRANT <privilege list> ON <table | view>
   TO <user list>
```

<user list>为：用户ID；public，允许所有有效用户拥有授予的权限；一个role

授予视图的权限并不意味着授予底层关系的任何权限。

权限的授予这必须已经拥有指定项的权限。

- select:允许对关系进行读访问，或使用视图进行查询的能力
- insert:插入元组的能力
- update:使用SQL更新语句进行更新的能力
- delete:删除元组的能力
- references:在创建关系时声明外键的能力
- all privilege:用作所有允许权限的简写形式
- all

带有授予选项：允许被授予权限的用户将该权限传递给其它用户。

Role:通过创建相应的"Role"，可以仅指定一次允许某类用户拥有通用权限的角色。

权限可以像授予或撤销用户权限一样授予或撤销角色的权限；角色可以分配给用户，甚至可以分配给其它角色。

撤销语句用于撤销授权：

```
REVOKE <privilege list> ON <table | view>
FROM <user list> [restrict | cascade]
```

从用户处撤销特权会导致其它用户也失去该特权，这被称为撤销级联。我们可以通过指定restrict来防止级联。

<privilege list>可以为all，用于撤销被撤销者可能持有的所有权限。如果<revoke-list>包含public，则除了那些被明确授予该权限的用户外，所有用户都将失去该权限。

如果同一授予者多次向同一用户授予相同的权限，则该用户在撤销操作后可能仍保留该

权限。

所有依赖于被撤销权限的权限也将被撤销。

审计跟踪是数据库所有更改的日志，其中包含诸如执行更改的用户以及更改执行时间等信息。用于跟踪错误/欺诈性更改。可以用trigger实现，但许多数据库系统直接提供。

4 嵌入式SQL

Embedded SQL是把SQL语句嵌入到“宿主语言”中的一种标准机制。

为什么要用？

- SQL功能不完备。例如流程控制、复杂计算、资源管理(文件、网络)等，要在宿主语言里处理更方便。
- 动态参数传递：通过宿主语言变量，可以在运行时构造查询条件并把结果带回程序
- 事务与业务逻辑结合：在同一程序中既能执行业务算法，又能做数据库操作。

1.Pre-processing:嵌入式SQL代码需要先由“预编译器”（如 pro*c、esql、nsqlprep 等）扫描，把所有 EXEC SQL ... END_EXEC 块中的 SQL 提取出来，生成普通函数调用（OCI、ODBC、CLI、LIBPQ 等），并保留宿主语言变量接口。预编译器还会插入头文件（如 #include <sqlca.h>）、定义 SQL 通讯区 SQLCA，用于保存每条SQL执行后的状态码（如 sqlca.sqlcode/sqlca.sqlerrd）。

2.DECLARE section

```
    EXEC SQL BEGIN DECLARE SECTION;
    char V_an[20], bn[20];
    float bal;
  EXEC SQL END DECLARE SECTION;
```

3.Single-Row Query:

```
    scanf("%s", V_an);
    EXEC SQL
      SELECT branch_name, balance
        INTO :bn, :bal
        FROM account
       WHERE account_number = :V_an;
    END_EXEC;
```

4.Cursor:

```
    EXEC SQL
      DECLARE c CURSOR FOR
        SELECT B.customer_name, B.customer_city
          FROM depositor D, customer B, account A
         WHERE D.customer_name = B.customer_name
           AND D.account_number = A.account_number
           AND A.balance > :v_amount;
    END_EXEC;
    EXEC SQL OPEN c END_EXEC;
    while (1) {
      EXEC SQL FETCH c INTO :cn, :cc END_EXEC;
      if (sqlca.sqlcode == 100) break;
      printf("customer:%s, city:%s\n", cn, cc);
    }
    EXEC SQL CLOSE c END_EXEC;
```

5.单行更新:

```
    scanf("%f", &delta);
    EXEC SQL
      UPDATE account
         SET balance = balance + :delta
       WHERE account_number = :V_an;
    END_EXEC;
```

6.多行更新:

```
    EXEC SQL
    DECLARE csr CURSOR FOR
      SELECT * FROM account
       WHERE branch_name = 'Perryridge'
       FOR UPDATE OF balance;
  END_EXEC;

  EXEC SQL OPEN csr;
  while (1) {
    EXEC SQL FETCH csr INTO :an, :bn, :bal;
    if (sqlca.sqlcode == 100) break;
    EXEC SQL
      UPDATE account
```

```
        SET balance = balance + 100
      WHERE CURRENT OF csr;
  END_EXEC;
}
EXEC SQL CLOSE csr;
```

5 ODBC和JDBC

5.1 ODBC

1.概念与定位

- ODBC（开放数据库互连）是一套由微软发起并被各大数据库厂商支持的 C API 标准，旨在为应用程序和各种关系数据库服务器之间提供统一、与具体 DBMS 无关的连接方式。
- 它不需要预编译，也不特定于某一个数据库。常见的 GUI 应用、电子表格、报表工具、脚本程序等都可以通过 ODBC 驱动访问任意支持 ODBC 的后端。

2.架构与组件

- ODBC 驱动管理器（Driver Manager）：驻留在客户端，负责加载不同数据库厂商提供的 ODBC 驱动并转发 API 调用。
- ODBC 驱动（Driver）：每个数据库（Oracle、SQL Server、MySQL、PostgreSQL 等）都提供相应驱动，实现标准 ODBC 接口并与其本地客户端库对接。
- 数据源名称（DSN, Data Source Name）：在操作系统的“ODBC 数据源管理器”里预先配置，记录数据库类型、服务器地址、端口、库名、用户名、密码等连接信息。应用只需引用 DSN 即可连向对应数据库。

```
SQLHENV henv;
SQLAllocEnv(&henv);

SQLHDBC hdbc;
SQLAllocConnect(henv, &hdbc);

SQLConnect(hdbc,
           "MyDSN", SQL_NTS,
           "user",  SQL_NTS,
           "pwd",   SQL_NTS);
```

```
11
12      SQLHSTMT hstmt;
13      SQLAllocStmt(hdbc, &hstmt);
14
15      SQLExecDirect(hstmt,
16                    "SELECT branch_name, SUM(balance) FROM account ↩
                         ↪GROUP BY branch_name",
17                    SQL_NTS);
18
19      SQLPrepare(hstmt, "UPDATE account SET balance = balance + ? ↩
            ↪WHERE account_number = ?", SQL_NTS);
20
21      SQLExecute(hstmt);
22
23      char branch[30];
24      double bal;
25      SQLBindCol(hstmt, 1, SQL_C_CHAR, branch, sizeof(branch), NULL↩
            ↪);
26      SQLBindCol(hstmt, 2, SQL_C_DOUBLE, &bal, 0, NULL);
27
28      while (SQLFetch(hstmt) == SQL_SUCCESS) {
29          printf();
30      }
31
32      SQLFreeStmt(hstmt, SQL_DROP);
33      SQLDisconnect(hdbc);
34      SQLFreeConnect(hdbc);
35      SQLFreeEnv(henv);
```

5.2 IDBC

1.概念与定位

- JDBC 是 Java 平台下访问关系数据库的官方 API 标准，定义在 java.sql 包中。与 ODBC 类似，它为 Java 应用提供统一的编程接口，后端具体数据库由相应的 JDBC 驱动实现。
- JDBC 支持查询 / 更新、参数化预编译语句、结果集（ResultSet）、事务控制、数据库元数据检索等功能。

2.编程模型与关键类

1. 加载驱动 Class.forName("oracle.jdbc.driver.OracleDriver");或者在新版 JDBC 中可不显式加载，依赖 SPI 机制。
2. 建立连接:

```
Connection conn = DriverManager.getConnection(
    "jdbc:oracle:thin:@hostname:1521:orcl",
    "user", "password"
);
```

3. 创建语句:Statement stmt = conn.createStatement();用于执行静态 SQL

```
PreparedStatement ps = conn.prepareStatement(
     "INSERT INTO account VALUES (?, ?, ?)"
);
ps.setString(1, "A_9732");
ps.setString(2, "Perryridge");
ps.setInt(3, 1200);
```

4. 执行 SQL

```
ResultSet rs = stmt.executeQuery(
"SELECT branch_name, AVG(balance) FROM account GROUP
   BY branch_name");
int count = stmt.executeUpdate(
    "UPDATE account SET balance = balance + 100 WHERE
       account_number = 'A_9732'"
);
```

5. 遍历结果

```
while (rs.next()) {
    String branch = rs.getString("branch_name");
    double avgBal = rs.getDouble(2);
    System.out.printf("%s: %.2f%n", branch, avgBal);
}
```

6. 异常处理:JDBC 通过抛出 SQLException 来报告各种数据库错误，应用应在 catch 块中检查 sqlState、errorCode 等。
7. 关闭资源:

```
rs.close();
```

```
        stmt.close();
        conn.close();
```

特性	ODBC	JDBC
平台/语言	C/C++（也可在其它语言中封装）	Java
驱动加载	需在客户端安装 ODBC 驱动并配置 DSN	通过 JDBC Driver 管理，通常添加 JAR
预编译/参数化	`SQLPrepare` / `SQLExecute`	`PreparedStatement`
结果集遍历	`SQLBindCol` + `SQLFetch`	`ResultSet.next()` + `getXXX()`
异常处理	返回码 + C 函数返回值	`SQLException` 抛出
元数据获取	`SQLTables`、`SQLColumns` 等 API	`DatabaseMetaData` 对象
易用性	较底层，需要手动管理句柄和内存	面向对象，跟 Java 语言集成度高

Figure 1