DB2 SQL 精萃

尚波

2010-5-30

前言

为什么写本书：

我是一名 Java 程序员，项目中使用 DB2，在过去的两年中，我几乎天天在写 SQL，而且都是一些很无聊的 SQL，不写又不行，因为这就是我的工作。既然不能改变现实，那就改变自己吧，我开始系统的学习 SQL 语句（当然，在这之前我认为 SQL 语句实在是没有什么可学的），随着学习的深入，我才发现，原来 SQL 语句其实并不简单，简单只是它的表面，有好多细节需要掌握，否则，稍不留神就会写出有逻辑错误的语句，更可怕的是，在海量数据库中，发现有逻辑错误的语句是非常难的，甚至是不可能的。还有一些本来用一条语句可以搞定的事情，有些人却在程序中写了一个循环语句来访问数据库（连接数据库是非常耗时的，尽管目前的软件都采用连接池），造成数据库的巨大压力。还有些人写 SQL 语句的时候根本不考虑索引，认为那是 DBA 事情。

当然我也犯过很多不可饶恕的错误，感谢我的领导 a9 和贾伟，以及其他同事对我的宽容。这本书是我在犯了无数错误后总结出来的，能看到本书，你是幸运的，如果你也在使用 DB2，那么本书不可错过。

声明：

联系作者：

E-mail: wave0409@163.com

博　客　：　http://blog.csdn.net/shangboerds　或　http://iamwave.javaeye.com/

目录

第一部分 DB2 SQL 精萃

第二部分 SQL PL 简介

第一部分 DB2 SQL 精萃

凡是知道数据库的人都知道 SQL，凡是对 SQL 有一点了解的人都觉得 SQL 很简单，凡是有这种感觉的人都是 SQL 得初级用户，因为他学会了增查删改就以为这就是 SQL 的全部。目前的大部分应用软件都是以数据库为中心，随着软件的运行，数据量会越来越大。如何用简洁、高效的 SQL 语句操作数据显得越来越重要。本部分系列文章将给大家介绍常用的一些 SQL 技巧。

连接字符串

如何将两个或多个字符串连接起来呢？DB2 提供了两种方法:

方法 1（利用运算符）：

```
VALUES 'A' || 'B';
```

方法 2（利用函数）：

```
VALUES CONCAT('A', 'B');
```

一般来说，当要连接多个字符串的时候，使用运算符要简单一点。

在字符串中输入单引号

字符串是用单引号括起来的，如果想在字符串输入单引号该怎么办呢？答案是用两个单引号代表一个单引号，如下 SQL 所示：

```
VALUES 'hello, i''m Scott'
```

在字符串中输入回车换行或其它特殊字符

很多人搞不清楚到底什么是回车（carriage return），什么是换行（line feed），下面简要介绍一下这两个概念的来历和区别。

在计算机还没有出现之前，有一种叫做电传打字机的玩意，每秒钟可以打 10 个字符。但是它有一个问题，就是打完一行换行的时候，要用去 0.2 秒，正好可以打两个字符。要是在这 0.2 秒里面，又有新的字符传过来，那么这个字符将丢失。于是，研制人员想了个办法解决这个问题，就是在每行后面加两个表示结束的字符。一个叫做"回车"，告诉打字机把打印头定位在左边界；另一个叫做"换行"，告诉打字机把纸向下移一行。这就是"换行"和"回车"的来历，从它们的英语名字上也可以看出一二。

后来，计算机发明了，这两个概念也就被般到了计算机上。那时，存储器很贵，一些科学家认为在每行结尾加两个字符太浪费了，加一个就可以。于是，就出现了分歧。Unix 系统里，每行结尾只有"<换行>"，即"\n"；Windows 系统里面，每行结尾是"<换行><回车>"，即"\n\r"；Mac 系统里，每行结尾是"<回车>"。一个直接后果是，Unix/Mac 系统下的文件在 Windows 里打开的话，所有文字会变成一行；而 Windows 里的文件在 Unix/Mac 下打开的话，在每行的结尾可能会多出一个^M 符号。

那么，如果我们要在数据库中存储某段文字，而这段文字包含换行和回车，该怎么办呢？请看下面的代码：

```
   VALUES 'Hello everyone' ||  CHR(13) || CHR(10) || 'i''
m Scott'
```

上面的 **CHR 函数**的作用是将 ASCII 码转化为字符，换行符的 ASCII 码是 10，回车符的 ASCII 码是 13。不只是回车和换行，如果你想输入其他任何特殊字符，你都可以采用上面的方式，用 **CHR 函数**进行转换。

DB2 INSERT 语句

有点 SQL 基础的人都会写 INSERT 语句，可是有很大一部分人不知道 DB2 的 INSERT 语句有**三种格式**，即:**一次插入一行,一次插入多行和从 SELECT 语句中插入。**考虑下面的情况：

```
1. CREATE TABLE USER
2. (
3. NAME VARCHAR(20) NOT NULL,---姓名
4. BIRTHDAY DATE---生日
5. );
```

现在要求你插入一行数据，我们这么写：

```
1. INSERT INTO USER (NAME,BIRTHDAY) VALUES ('张三','2000-1-1');
```

现在要求你插入三行数据，我们这么写：

```
1. INSERT INTO USER (NAME,BIRTHDAY) VALUES ('张三','2000-1-1');
2. INSERT INTO USER (NAME,BIRTHDAY) VALUES ('李四','2000-1-1');
3. INSERT INTO USER (NAME,BIRTHDAY) VALUES ('王五','2000-1-1');
```

除此之外，我们还可以这么写：

```
1. INSERT INTO USER (NAME,BIRTHDAY) VALUES
2. ('张三','2000-1-1'),
3. ('李四','2000-1-1'),
4. ('王五','2000-1-1');
```

那么后一种写法有什么好处呢？有两点好处：

1、性能更好。

2、由于一条语句，所以它们是一个处理单元，要么都插入，要么都不插入。

除此之外，我们还可以从 SELECT 中插入，格式如下：

```
1. INSERT INTO USER (NAME,BIRTHDAY)
2. SELECT <COLUMN1>,<COLUMN2> FROM <TABLE_NAME> WHERE ...
```

以上比较简单，我就不举例子了。

DB2 UPDATE 语句

我估计，只要是知道 SQL 语句的人都会用 UPDATE 语句，可是大部分人不知道 UPDATE 语句有两种写法，首先，考虑下面的情况：

```
1. CREATE TABLE STUDENT
2. (
3.     ID INT,---学号
4.     NAME VARCHAR(20) NOT NULL,---姓名
5.     BIRTHDAY DATE---生日
6.     primary key(ID)
7. );
8.
9.
10.INSERT INTO STUDENT (ID,NAME,BIRTHDAY) VALUES
11.(1,'张三','1991-1-1'),
12.(2,'李四','1991-1-1'),
13.(3,'王五','1990-1-1');
```

假设让你更新一下张三的生日，很简单，我们可以这么写：

```
1. UPDATE STUDENT SET BIRTHDAY='1991-1-5' WHERE NAME='张三';
```

除此之外，我们还可以这么写：

```
1. UPDATE
2. (
3.     SELECT * FROM STUDENT WHERE NAME='张三'
4. )
5. SET BIRTHDAY='1991-1-5'
```

就上面这个例子而言，通常我们不会使用第二种写法，因为这种写法可读性不如第一种写法好，下面我们举一个用第一种方法办不到得例子，可是用第二种方法却非常简单（这是一个生产环境实际的例子，通常用在银行中，关于表定义，我做了简化），考虑如下情况：

```
1. CREATE TABLE TRANSACTION
2. (
3.     CUSTOMERID VARCHAR(10),---顾客号
4.     SEQ INT NOT NULL,---流水号(每个顾客从 1 开始)
5.     PROCESSDATE DATE,---处理日
6.     AMOUNT DECIMAL(16,4)---金额
7. );
```

对于上面表的**流水号(SEQ)字段**，每个顾客从 1 开始，而且它的顺序，其实就是**处理日(PROCESSDATE)**排序后的顺序。细心的朋友可能已经发现，这样设计明显违反了表的第二范式，造成数据冗余。确实是这样的，为什么要这样设计呢？真实的原因我也不知道，只能问该系统的设计者。我猜测可能的原因是，系统的设计者**将银行用的纸质报表直接转化为了数据库中的表**，纸质报表有流水号(SEQ)字段是可以的，因为纸质报表我们无法排序，也不可能让**会计**去数每个顾客到底发生了多少交易，但是在数据库的表中，该字段则完全没有必要，因为我们可以通过对处理日(PROCESSDATE)排序后产生该结果。相反，如果设置流水号(SEQ)字段，就有可能因为这样那样的问题**导致流水号(SEQ)和处理日(PROCESSDATE)排序结果不一致**，这时候就要求我们更新流水号(SEQ)，那么，我们该怎么更新呢？这个问题很好解决，通常我们会将查询结果排序后，更新其中的每一条记录。这样做是可以的，但是有点笨，我们能不能用一条语句来更新呢？答案是可以的，如下：

```
1. UPDATE
2. (
3.     SELECT
4.         TT.*,
5.         ROW_NUMBER() OVER() AS RN
6.     FROM
7.         TRANSACTION AS TT WHERE CUSTOMERID=...
8. )
9. SET SEQ=RN
```

怎么样，是不是很简单。通过以上的分析，你可能对 UPDATE 语句的两种方法有一定的了解。如果你对上面语句的 ROW_NUMBER() OVER()还不熟悉，请参见：**DB2 在线分析处理（OLAP 函数的使用）**

DB2 中删除数据

大家对如何删除数据都不陌生，我们习惯性的这么写：

```
1. DELETE FROM <TABLE_NAME> WHERE <CONDITION>;
```

其实这么写性能并不好，尤其是删除大量数据的时候，要想获得更好的性能，可以采用如下方式：

```
1. DELETE FROM
2. (
3. SELECT * FROM <TABLE_NAME> WHERE <CONDITION>
4. );
```

那如果要把一个表的所有数据都删除了，该怎么办？有人可能会说，这简单啊，把 WHERE 子句去掉不就可以了。回答正确，**这是一种方法**。当数据量很大时，删除数据需要很长时间，有人可能会采用先 DROP TABLE，然后 CREATE TABLE 的方式，**这是第二种方法。**这样处理虽然很快，但是比较麻烦。其实还有**第三种方法，**更快、更简单，如下：

```
1. ALTER TABLE <TABLE_NAME> ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABL
   E
```

郑重申明，使用以上语句后，对表的操作将不会记录日志，恢复的时候可能有问题，所以特别适合临时表。以上语句要慎重使用，出现任何后果本人概不负责。当然，还有其他方法，但那**不是 SQL 了，是 DB2 的命令**，我们这里就不介绍了。

相关子查询

我们先来看两个表的定义：

```
--用户
CREATE TABLE USER
(
USERID INTEGER NOT NULL,---用户 ID
COMPANYID INTEGER,---公司 ID
TELNO VARCHAR(12)---用户电话
);

--公司
CREATE TABLE COMPANY
(
COMPANYID INTEGER NOT NULL,---公司 ID
TELNO VARCHAR(12)---公司电话
);
```

大家对子查询都非常熟悉，可是我发现，很多人都不知道子查询有两种格式：一种是相关子查询（Correlated Sub-Query），另一种是非相关子查询（Uncorrelated Sub-Query）。下面我们通过一个例子来对比一下这两种子查询的不同。假设现在让你查询一下公司电话是 88888888 的用户有哪些，我们可以使用如下语句：

```
--非相关子查询（Uncorrelated Sub-Query）
SELECT * FROM USER WHERE COMPANYID IN
(
SELECT COMPANYID FROM COMPANY WHERE TELNO='88888888'
);

--相关子查询（Correlated Sub-Query）
SELECT * FROM USER AS U WHERE EXISTS
(
SELECT * FROM COMPANY  AS C WHERE TELNO='88888888' AND U.COMPANYID=C.C
OMPANYID
);
```

以上两条语句的用作是相同的，对比后我们发现，相关子查询的子句（也就是括号中的语句：ELECT * FROM COMPANY AS C WHERE TELNO='88888888' AND

U.COMPANYID=C.COMPANYID）依赖外部语句的条件，不能单独执行；而非相关子查询的子句是可以单独执行的。
就以上这个例子来说，我们使用相关子查询无论从性能和可读性都不如非相关子查询，下面我们来看一个使用非相关子查询办不到的例子，假设现在让你把用户电话更新成公司电话，怎么办？有些人可能采用如下的方式构造 update sql，然后执行，如下：

```
1. SELECT 'UPDATE USER SET TELNO=''' || TELNO || ''' WHERE COMPANYID=' ||
    CHAR(COMPANYID) || ';' FROM COMPANY
```

这么做是可以的，但是有点笨（当然，自己觉得挺聪明，因为他可能觉得自己使用了别人不常用或不知道的 using sql to make sql），我们还有更好的方法，就是采用相关子查询，如下所示：

```
1. UPDATE USER AS U SET TELNO=
2. (
3. SELECT TELNO FROM COMPANY AS C WHERE U.COMPANYID=C.COMPANYID
4. );
```

以上就是相关子查询的一种用途，还有好多其他用途等待你去挖掘。

多字段查询

假设现在有如下表：

```
CREATE TABLE USER
(
NAME VARCHAR(20) NOT NULL,---姓名
BIRTHDAY DATE---生日
);
```

现在，要求你把 生日是 1980-8-8，学历是 硕士的人找出来，很简单，我们可以这么写：

```
SELECT * FROM USER WHERE BIRTHDAY='1980-8-8' AND DEGREE='硕士';
```

我们还可以这么写：

```
SELECT * FROM USER WHERE (BIRTHDAY,DEGREE)=('1980-8-8','硕士');
```

这一点在需要多个条件查找表的时候，非常有用，如下：

```
SELECT * FROM USER WHERE (BIRTHDAY,DEGREE) IN
(
---此处是子查询
);
```

不仅如此，在更新的时候，也非常有用，如下：

```
UPDATE USER SET (BIRTHDAY,DEGREE)=
(
---此处是相关子查询
)
WHERE <CONDITION>;
```

在 ORDER BY 子句中加入主键或唯一键

ORDER BY 子句非常简单，正因为简单，很多人不太在意，随意使用 ORDER BY 子句很可能出现逻辑错误，考虑如下情况：

```
1. CREATE TABLE STUDENT
2. (
3.     ID INT,---学号
4.     NAME VARCHAR(20) NOT NULL,---姓名
5.     BIRTHDAY DATE---生日
6.     primary key(ID)
7. );
8.
9.
10. INSERT INTO STUDENT (ID,NAME,BIRTHDAY) VALUES
11. (1,'张三','1991-1-1'),
12. (2,'李四','1991-1-1'),
13. (3,'王五','1990-1-1');
```

假设让你查找一下年龄最小的学生，我们很自然的会写出如下 SQL：

```
1. SELECT * FROM STUDENT
2. ORDER BY BIRTHDAY
3. FETCH FIRST 1 ROW ONLY
```

遗憾的是，上面的语句并不总是正确的，因为张三和李四的年龄是相同的，我们应该把学号(ID)添加在 ORDER BY 子句中，正确的 SQL 如下：

```
1. SELECT * FROM STUDENT
2. ORDER BY BIRTHDAY, ID
3. FETCH FIRST 1 ROW ONLY
```

只要是 ORDER BY 子句中没有主键或唯一键，就有可能出现上面的情况，所以**作为一条规则，我们应该在 ORDER BY 子句中加入主键或唯一键。**

GROUPING SETS、ROLLUP、CUBE

大家对 GROUP BY 应该比较熟悉，如果你感觉自己并不完全理解 GROUP BY，那么本文不适合你。还记得当初学习 SQL 的时候，总是理解不了 GROUP BY 的作用，经过好长时间才终于明白 GROUP BY 的真谛。当然，这和我本人笨也有关系，但是 GROUP BY 的确不好理解。本文将介绍 DB2 GROUPING SETS、ROLLUP、CUBE 的使用方法，这些关键字比 GROUP BY 更难理解，所以阅读本文的时候，一定要慢，仔细的分析，你理解的越多，需要记忆的就越少。

我们首先来看 GROUPING SETS 的使用方法，请看下面的例子

```
1. GROUP BY GROUPING SETS (A,B,C)  等价与  GROUP BY A
2.                                          UNION ALL
3.                                          GROUP BY B
4.                                          UNION ALL
5.                                          GROUP BY C
```

从字面上理解，GROUPING SETS 就是 GROUP 集合的意思，确实是这样的，从上面的例子，我们可以很容易的理解 GROUPING SETS 的使用方法，但是使用括号的时候需要我们特别注意，请看下面的例子

```
1. GROUP BY GROUPING SETS ((A,B,C))  等价与  GROUP BY A,B,C
2.
3.
4. GROUP BY GROUPING SETS (A,(B,C))  等价与  GROUP BY A
5.                                            UNION ALL
6.                                            GROUP BY B,C
```

我们应该把**括号里面的所有内容看做一个整体**，这个整体必须在同一个 GROUP BY 语句中，例如，语句 2 中的 B,C 在括号中，B,C 必须在同一个 GROUP BY 语句中，千万别把他们拆开，写出 GROUP BY B UNION ALL GROUP BY C，那样就大错特错了。

我们还可以在一个 GROUP BY 语句中多次使用 GROUPING SETS，如下：

```
1. GROUP BY GROUPING SETS (A)  等价于  GROUP BY A,B,C
2.          ,GROUPING SETS (B)
3.          ,GROUPING SETS (C)
4.
5.
6. GROUP BY GROUPING SETS (A)  等价于  GROUP BY A,B,C
7.          ,GROUPING SETS ((B,C))
8.
```

```
9.
10.GROUP BY GROUPING SETS (A)   等价于   GROUP BY A,B
11.         ,GROUPING SETS (B,C)          UNION ALL
12.                                       GROUP BY A,C
```

我们还可以混合使用，如下：

```
1.  GROUP BY A                        等价于   GROUP BY A
2.           ,B                                          ,B
3.           ,GROUPING SETS ((B,C))                      ,C
4.
5.
6.  GROUP BY A                      等价于   GROUP BY A,B,C
7.           ,B                               UNION ALL
8.           ,GROUPING SETS (B,C)             GROUP BY A,B
9.
10.
11.GROUP BY A                      等价于 GROUP BY A,B,C
12.         ,B                             UNION ALL
13.         ,C                             GROUP BY A,B,C
14.         ,GROUPING SETS (B,C)
```

请特别注意上面的第 3 条语句。

下面我们介绍一下 ROLLUP 和 CUBE 关键字，它们的使用方式类似，作用也类似，都是用来为 GROUP BY 语句返回的结果添加汇总信息，也可以说，它们是对分组结果进行二次分组。下面我们看一个简单的例子，如下：

```
1.  SELECT
2.      DEPT    AS 部门,
3.      SEX     AS 性别,
4.      AVG(SALARY) AS 平均工资
5.  FROM
6.  (
7.      --姓名   性别   部门   工资
8.      VALUES
9.      ('张三','男','市场部',4000),
10.     ('赵红','男','技术部',2000),
11.     ('李四','男','市场部',5000),
12.     ('李白','女','技术部',5000),
13.     ('王五','女','市场部',3000),
14.     ('王蓝','女','技术部',4000)
15.) AS EMPLOY(NAME,SEX,DEPT,SALARY)
16.GROUP BY ROLLUP(DEPT,SEX)
```

```
17. ORDER BY 部门,性别
18.
19.
20. 查询结果：
21. 部门            性别          平均工资
22. 市场部           女            3000
23. 市场部           男            4500
24. 市场部           NULL          4000
25. 技术部           女            4500
26. 技术部           男            2000
27. 技术部           NULL          3666
28. NULL             NULL          3833
```

值得注意的是，上面的 ROLLUP 语句中，部门(DEPT)和性别(SEX)的顺序非常重要，如果我们互换一下它两的顺序，将得到不同的结果，如下：

```
1.  SELECT
2.      SEX     AS 性别,
3.      DEPT    AS 部门,
4.      AVG(SALARY) AS 平均工资
5.  FROM
6.  (
7.      --姓名   性别   部门   工资
8.      VALUES
9.      ('张三','男','市场部',4000),
10.     ('赵红','男','技术部',2000),
11.     ('李四','男','市场部',5000),
12.     ('李白','女','技术部',5000),
13.     ('王五','女','市场部',3000),
14.     ('王蓝','女','技术部',4000)
15. ) AS EMPLOY(NAME,SEX,DEPT,SALARY)
16. GROUP BY ROLLUP(SEX,DEPT)
17. ORDER BY 性别,部门
18.
19.
20. 查询结果：
21. 性别            部门          平均工资
22. 女            市场部          3000
23. 女            技术部          4500
24. 女            NULL            4000
25. 男            市场部          4500
26. 男            技术部          2000
27. 男            NULL            3666
28. NULL          NULL            3833
```

CUBE 语句比 ROLLUP 语句返回更多的内容，以下是将上面语句的 ROLLUP 替换为 CUBE 后得到的结果：

```
SELECT
    DEPT   AS 部门,
     SEX    AS 性别,
     AVG(SALARY) AS 平均工资
FROM
(
     --姓名   性别   部门   工资
     VALUES
     ('张三','男','市场部',4000),
     ('赵红','男','技术部',2000),
     ('李四','男','市场部',5000),
     ('李白','女','技术部',5000),
     ('王五','女','市场部',3000),
     ('王蓝','女','技术部',4000)
) AS EMPLOY(NAME,SEX,DEPT,SALARY)
GROUP BY CUBE(DEPT,SEX)
ORDER BY 部门,性别
```

查询结果：

部门	性别	平均工资
市场部	女	3000
市场部	男	4500
市场部	NULL	4000
技术部	女	4500
技术部	男	2000
技术部	NULL	3666
NULL	女	4000
NULL	男	3666
NULL	NULL	3833

如果我们替换 CUBE 语句中部门(DEPT)和性别(SEX)的顺序，我们将会得到相同的结果。

SOME，ANY，All，EXISTS，IN

这几个关键字有一个共同点，那就是它们一般应用于子查询中。大家对 IN 都比较熟悉，这里我们就不介绍了，下面我们看一看其他几个关键字的使用，首先，我们定义如下表：

```
--学生
CREATE TABLE STUDENT
(
ID VARCHAR(8),---学号
NAME VARCHAR(20),---姓名
CLASS VARCHAR(20),---班级
CHINESE FLOAT,---语文成绩
MATH FLOAT---数学成绩
);


INSERT INTO STUDENT (ID, NAME, CLASS, CHINESE, MATH) VALUES
('20090001', '张三', '五年级 A 班', 80 ,90),
('20090002', '李四', '五年级 A 班', 60 ,75),
('20090003', '王五', '五年级 A 班', 90 ,95),
('20090004', '赵红', '五年级 B 班', 70 ,90),
('20090004', '李白', '五年级 B 班', 85 ,80),
('20090005', '王蓝', '五年级 B 班', NULL ,70);
```

假设现在让你查询一下，A 班哪些学生的数学成绩高于 B 班数学成绩的**最小值**，怎么办？我们可以采用如下 SQL：

```
SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND MATH >
(
    SELECT MIN(MATH) FROM STUDENT WHERE CLASS='五年级 B 班'
);
```

除此之外，我们还可以使用 **SOME 或 ANY**。注意：ANY 和 SOME 的作用完全和使用方式完全相同，不知道制定 SQL 标准的人为什么要定义两个关键字。我们来看一下以上问题通过 SOME 或 ANY 怎么实现。如下 SQL 所示：

```
SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND MATH > ANY
(
```

```
3.      SELECT MATH FROM STUDENT WHERE CLASS='五年级 B 班'
4. );
```

假设现在让你查询一下，A 班哪些学生的数学成绩高于 B 班数学成绩的**最大值**，怎么办呢？我们可以采用如下 SQL：

```
1. SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND MATH >
2. (
3.      SELECT MAX(MATH) FROM STUDENT WHERE CLASS='五年级 B 班'
4. );
```

除此之外，我们还可以**使用 ALL**,如下 SQL 所示：

```
1. SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND MATH > ALL
2. (
3.      SELECT MATH FROM STUDENT WHERE CLASS='五年级 B 班'
4. );
```

至此，我们可以总结出 SOME,ANY,ALL 和 MIN, MAX 的**对应关系**：

```
1. > ANY(sub-qurey)   ---   > MIN(sub-qurey)
2. < ANY(sub-query)   ---   < MAX(sub-qurey)
3. > ALL(sub-query)   ---   > MAX(sub-qurey)
4. < ALL(sub-query)   ---   < MIN(sub-qurey)
```

至此，你应该理解了 SOME,ANY,ALL 关键字的作用了吧。下面我们看一看 **EXISTS 关键字**的作用。EXISTS 的作用比较简单，它只关注它后面的子查询返没返回值，而不在乎返回多少。如果返回，则整个表达式就为真，否则为假。**NOT EXISTS 关键字则和 EXISTS 作用相反**。假设现在让你查询一下有没有数学成绩为 100 的学生，如果有，则将所有学生的数学成绩输出，如果没有，则什么都不输出，我们使用 EXISTS 实现，如下：

```
1. SELECT NAME,MATH FROM STUDENT WHERE EXISTS
2. (
3.      SELECT * FROM STUDENT WHERE MATH=100
4. );
```

至此，以上几个关键字的作用全部介绍给大家了，不知大家理解了没有。以上操作都针对数学成绩，如果你认为对语文成绩也执行类似的操作会得到类似的答案的话，那么你就错了。一切的原因都是因为 NULL 引起的 。下面，我们来讨论 NULL 对各个关键字的影响。

```
---语句 1
SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND CHINESE >
(
    SELECT MAX(CHINESE) FROM STUDENT WHERE CLASS='五年级 B 班'
);

---语句 2
SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND CHINESE > ALL

(
    SELECT CHINESE FROM STUDENT WHERE CLASS='五年级 B 班'
);
```

通常，我们认为语句 1 和语句 2 会返回同样的结果，然而上面两条语句返回的结果却令人吃惊，语句 1 返回王五，语句 2 则什么也没返回，为什么会出现这样的情况呢？MAX 函数默认会忽略 NULL 值，所以语句 1 返回了王五。那么为什么语句 2 返回 NULL 呢？答案是我也不知道。哪位朋友知道的话请告诉我一下。不仅如此，下面两条语句也返回不同的结果：

```
---语句 1
SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND CHINESE <
(
    SELECT MIN(CHINESE) FROM STUDENT WHERE CLASS='五年级 B 班'
);

---语句 2
SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND CHINESE < ALL

(
    SELECT CHINESE FROM STUDENT WHERE CLASS='五年级 B 班'
);
```

不仅如此，试一试下面的语句（子查询没有返回任何记录）：

```
---语句 1
SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND CHINESE <
(
    SELECT MIN(CHINESE) FROM STUDENT WHERE 1<>1
);

--语句 2
```

```
8.  SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND CHINESE < ALL
9.  (
10.     SELECT CHINESE FROM STUDENT WHERE 1<>1
11. );
```

上面的几个例子提醒大家，**使用 ALL 的时候应该特别注意**，一不留神就会返回我们不期望的结果。
还有个关键字需要大家注意，那就是 **NOT IN**，请看下面的例子：

```
1.  SELECT NAME FROM STUDENT WHERE CLASS='五年级 A 班' AND CHINESE NOT IN
2.  (
3.     80,60,NULL
4.  );
```

如果你认为以上语句返回王五的话，那么，你就错了，虽然我们一般不会像上面那样主动写出 NULL 值，但是不能保证子查询也不会返回 NULL 值，所以在使用 NOT NULL 时也需要特别注意。

UNION, INTERSECT, EXCEPT

这几个关键字是用来操作集合的。**UNION** 用来求两个集合的并集，**INTERSECT** 用来求两个集合的交集，**EXCEPT** 用来求在第一个集合中存在，而在第二个集合中不存在的记录。每个关键字后面都可以接 ALL（**UNION ALL, INTERSECT ALL, EXCEPT ALL**），如果不接 ALL，操作集合将会去掉重复值，下面我们通过一个例子来对比一下它们直接的不同。

```
---UNION
VALUES ('A'), ('A'), ('B'), ('B'), ('C')
UNION
VALUES ('A'), ('B'), ('B'), ('D'), ('E')

---结果:
A
B
C
D
E


---UNION ALL
VALUES ('A'), ('A'), ('B'), ('B'), ('C')
UNION ALL
VALUES ('A'), ('B'), ('B'), ('D'), ('E')

---结果:
A
B
B
D
E
A
A
B
B
C


---INTERSECT
```

```
VALUES ('A'), ('A'), ('B'), ('B'), ('C')
INTERSECT
VALUES ('A'), ('B'), ('B'), ('D'), ('E')

---结果:
A
B


---INTERSECT ALL
VALUES ('A'), ('A'), ('B'), ('B'), ('C')
INTERSECT ALL
VALUES ('A'), ('B'), ('B'), ('D'), ('E')

---结果:
A
B
B


---EXCEPT
VALUES ('A'), ('A'), ('B'), ('B'), ('C')
EXCEPT
VALUES ('A'), ('B'), ('B'), ('D'), ('E')

---结果:
C


---EXCEPT ALL
VALUES ('A'), ('A'), ('B'), ('B'), ('C')
EXCEPT ALL
VALUES ('A'), ('B'), ('B'), ('D'), ('E')

---结果:
A
C
```

大家对比一下它们之间的结果就可以看出它们之间的区别，不过有两个问题需要注意:
1、UNION 和 INTERSECT 两别集合可以互换的，但是 EXCEPT 互换将有不同的结果，如下：

```
---语句 1
VALUES ('A'), ('A'), ('B'), ('B'), ('C')
```

```
3. EXCEPT
4. VALUES ('A'), ('B'), ('B'), ('D'), ('E')
5.
6. ---结果:
7. C
8.
9.
10. ---语句 2
11. VALUES ('A'), ('B'), ('B'), ('D'), ('E')
12. EXCEPT
13. VALUES ('A'), ('A'), ('B'), ('B'), ('C')
14.
15. ---结果:
16. D
17. E
```

2、注意它们之间的优先级，EXCEPT 的优先级要高于 UNION 和 INTERSECT，一般情况下如果多个关键字混合使用最好使用括号。

在操作数据的同时查看操作前或操作后的值

我们经常需要对数据库进行插入、更新、删除等操作，如果我们想在更新数据的同时查看一下更新后的值，该怎么办呢？怎么样？是不是不太明白我说的话，不要紧，考虑下面的情况：

```
1. CREATE TABLE USER
2. (
3. NAME VARCHAR(20) NOT NULL,--姓名
4. SALARY FLOAT NOT NULL--工资
5. );
6. INSERT INTO USER (NAME,SALARY) VALUES
7. ('张三', 1000),
8. ('李四', 2000),
9. ('王五', 2400),
10.('赵六', 2800),
11.('高七', 3000);
```

假设现在要给工资<=2000 的员工涨工资，在原来的基础上提供 20%，我们可以这么写：

```
1. UPDATE USER SET SALARY=SALARY*(1+0.2) WHERE SALARY<=2000
```

当你更新完上面的语句后，发现自己很冲动，没有查看有哪些员工就给更新了，然后你可能觉得不要紧，用如下语句查看一下：

```
1. SELECT * FROM USER WHERE SALARY<=2400
```

遗憾的是上面语句查询出来的结果已经不全是你更新的数据了，王五也被查询出来了，而王五并没有涨工资。那么，有没有一种方法，在更新的同时查看一下哪些值被更新了，更新前或更新后的值是多少呢？答案是有，如下：

```
1. SELECT * FROM FINAL TABLE
2. (
3.     UPDATE USER SET SALARY=SALARY*(1+0.2) WHERE SALARY<=2000
4. )
```

如果更新操作影响的行有很多，而我们只想看一下有没有姓李的人，只取 10 条查看一下，该怎么办呢？如下：

```
SELECT * FROM FINAL TABLE
(
    UPDATE USER SET SALARY=SALARY*(1+0.2) WHERE SALARY<=2000
)
 WHERE NAME LIKE '李%'
FETCH FIRST 10 ROWS ONLY;
```

看到这，如果你执行了上面的语句，那么你可能发现，以上语句的查询结果是更新后的值，那么，我们想查看更新前的值，该怎么办呢？很简单，我们只要把 **FINAL TABLE** 换成 **OLD ABLE** 就可以了，如下：

```
SELECT * FROM OLD TABLE
(
    UPDATE USER SET SALARY=SALARY*(1+0.2) WHERE SALARY<=2000
)
 WHERE NAME LIKE '李%'
FETCH FIRST 10 ROWS ONLY;
```

那么，我既想看一下更新前的值，又想看一下更新后的值，该怎么呢？是不是把关键字 **FINAL** 和 **OLD** 一起使用就可以了呢？答案没有直接的方法同时查看更新前和更新后的值，但是我们可以通过 INCLUDE 关键字，创造虚拟字段来包含更新前的值，如下：

```
SELECT * FROM FINAL TABLE
(
    UPDATE USER
    INCLUDE (OLD_SALARY FLOAT)
    SET SALARY=SALARY*(1+0.2),OLD_SALARY=SALARY
    WHERE SALARY<=2000
)
```

除了 **FINAL** 和 **OLD** 关键字外，其实还有个关键字：**NEW**，它的作用是，**查看 SQL 更新后，trigger（触发器）执行前的值。**
另外把上面的 UPDATE 语句换成 INSERT 和 DELETE 语句同样适用。
上面的这个例子不是很实用，其实这个功能最典型的用途是，当我们向自动生成主键的表插入数据后，我们很可能需要查看生成的主键值，这个时候，这个功能将相当有用。

DB2 Merge 语句的使用

DB2 Merge 语句的作用非常强大，它可以将一个表中的数据合并到另一个表中，在合并的同时可以进行插入、删除、更新等操作。我们还是先来看个简单的例子吧，假设你定义了一个雇员表（employe），一个经理表（manager），如下所示：

```
---雇员表（EMPLOYE）
CREATE TABLE EMPLOYE (
EMPLOYEID INTEGER  NOT NULL,---员工号
NAME VARCHAR(20) NOT NULL,---姓名
SALARY DOUBLE---薪水
);
INSERT INTO EMPLOYE (EMPLOYEID,NAME,SALARY) VALUES
(1,'张三',1000),
(2,'李四',2000),
(3,'王五',3000),
(4,'赵六',4000),
(5,'高七',5000);
--经理表（MANAGER）
CREATE TABLE MANAGER (
EMPLOYEID INTEGER  NOT NULL,---经理号
NAME VARCHAR(20) NOT NULL,---姓名
SALARY DOUBLE---薪水
);
INSERT INTO MANAGER (MANAGERID,NAME,SALARY) VALUES
(3,'王五',5000),
(4,'赵六',6000);
```

经过一段时间，你发现这样的数据模型，或者说表结构设计简直就是一大败笔，经理和雇员都是员工嘛，为什么要设计两个表呢？发现错误后就需要改正，所以你决定，删除经理表（MANAGER）表，将 MANAGER 表中的数据合并到 EMPLOYE 表中，仔细分析发现，王五在两个表中都存在（可能是干的好升官了），而刘八在 EMPLOYE 表中并不存在，现在，我们要求把 EMPLOYE 表中不存在的 MANAGER 都插入到 EMPLOYE 表中，存在的更新薪水。该怎么办呢？这个问题并不难，通常，我们可以分两步，如下所示：

```
--更新存在的
UPDATE EMPLOYE AS EM SET SALARY=(SELECT SALARY FROM MANAGER WHERE MANAGERID=EM.EMPLOYEID)
WHERE EMPLOYEID IN (
```

```
4.  SELECT MANAGERID FROM MANAGER
5.  );
6.  ---插入不存在的
7.  INSERT INTO EMPLOYE (EMPLOYEID,NAME,SALARY)
8.  SELECT MANAGERID,NAME,SALARY FROM MANAGER WHERE MANAGERID NOT IN (
9.  SELECT EMPLOYEID FROM EMPLOYE
10. );
```

上面的处理是可以的，但是我们还可以有更简单的方法，就是用 Merge 语句，如下所示：

```
1.  MERGE INTO EMPLOYE AS EM
2.  USING MANAGER AS MA
3.  ON EM.EMPLOYEID=MA.MANAGERID
4.  WHEN MATCHED THEN UPDATE SET EM.SALARY=MA.SALARY
5.  WHEN NOT MATCHED THEN INSERT VALUES (MA.MANAGERID,MA.NAME,MA.SALARY);
```

在上面的处理中，我们用经理表（MANAGER）的薪水更新了雇员表（EMPLOYE）的薪水，假设现在要求，如果经理表（MANAGER）的薪水>雇员表（EMPLOYE）的薪水的时候更新，否则不更新，怎么办呢？如下：

```
1.  MERGE INTO EMPLOYE AS EM
2.  USING MANAGER AS MA
3.  ON EM.EMPLOYEID=MA.MANAGERID
4.  WHEN MATCHED AND EM.SALARY<MA.SALARY THEN UPDATE SET EM.SALARY=MA.SALA
    RY
5.  WHEN NOT MATCHED THEN INSERT VALUES (MA.MANAGERID,MA.NAME,MA.SALARY);
```

不仔细的朋友可能没有看出上面两条语句的区别，哈哈，请仔细对比一下这两条语句。上面的语句中多了 ELSE IGNORE 语句，它的意思正如它英文的意思，其它情况忽略不处理。如果你认为理论上应该不存在 EM.SALARY>MA.SALARY 的数据，如果有，说明有问题，你想抛个异常，怎么办？如下：

```
1.  MERGE INTO EMPLOYE AS EM
2.  USING MANAGER AS MA
3.  ON EM.EMPLOYEID=MA.MANAGERID
4.  WHEN MATCHED AND EM.SALARY<MA.SALARY THEN UPDATE SET EM.SALARY=MA.SALA
    RY
5.  WHEN MATCHED AND EM.SALARY>MA.SALARY THEN SIGNAL SQLSTATE '70001' SET
     MESSAGE_TEXT = 'EM.SALARY>MA.SALARY'
```

```
6. WHEN NOT MATCHED THEN INSERT VALUES (MA.MANAGERID,MA.NAME,MA.SALARY)

7. ELSE IGNORE;
```

对于 EM.SALARY>MA.SALARY 的情况，如果你不想抛异常，而是删除 EMPLOYE 中的数据，怎么办？如下：

```
1. MERGE INTO EMPLOYE AS EM
2. USING MANAGER AS MA
3. ON EM.EMPLOYEID=MA.MANAGERID
4. WHEN MATCHED AND EM.SALARY<MA.SALARY THEN UPDATE SET EM.SALARY=MA.SALA
   RY
5. WHEN MATCHED AND EM.SALARY>MA.SALARY THEN DELETE
6. WHEN NOT MATCHED THEN INSERT VALUES (MA.MANAGERID,MA.NAME,MA.SALARY)

7. ELSE IGNORE;
```

以上简单介绍了 Merge 语句的使用，它的应用不只是上面介绍的情况，其实它可以应用在很多其他语句不好处理情况，这需要你去发现，记住熟能生巧。

采集样本数据

我们经常会遇到这样的情况，想看看某个表中的若干条数据，如 10 条、20 条等。在 DB2 中，我们可以这么写：

```
1. SELECT * FROM <TABLE_NAME> FETCH FIRST 10 ROWS ONLY;
```

不知道你注意到没有，以上这条语句无论你执行多少遍，结果集是不变的。那么我想每次随机的查询 10 条记录看看，该怎么处理呢？可以使用下面的 SQL：

```
1. SELECT * FROM <TABLE_NAME> ORDER BY RAND() FETCH FIRST 10 ROWS ONLY ;
```

上面是最简单的采集样本数据的方法，在 DB2 中，更为专业的是使用 TABLESAMPLE 采集样本数据。那么，为什么要采集样本数据呢？主要原因是当我们对海量数据进行分组统计时，即费时又费力，这时候，我们可以采集样本数据，然后对样本数据进行统计，以预测整体趋势。

一：语法

```
1. SELECT ... FROM
2. <table-name> TABLESAMPLE [BERNOULLI | SYSTEM] (percent) REPEATABLE (num)
3. WHERE ...
```

二：示例

```
1. SELECT *
2. FROM staff TABLESAMPLE BERNOULLI(8) REPEATABLE(586)
3. ORDER BY id;
4.
5. --说明
6. 从 staff 表中，采用 BERNOULLI 抽样方法，抽取 8%的样本数据，REPEATABLE 表示多次执行相同的语句返回相同的结果。
```

三：采样方法

1、BERNOULLI(行级别伯努利采样)：它检查每一行，准确率高，但是性能差。

2、SYSTEM(系统页级采样)：它检查每一数据页(一个数据页包含若干行)，性能高，但准确率差。

IN 与 DISTINCT

在开始本文之前，我们先看条 SQL 语句，如下：

```
1. SELECT <field_name> FROM <...> WHERE <field_name> IN
2. (
3.     SELECT DISTINCT <field_name> FROM <...>
4. );
```

怎么样？看上去是不是很熟悉，可能你曾经写过 或者 看到别人写过这样的语句，或者压根就没注意。上述语句想表达的意思是：去**掉结果集中的重复值**，然而却**画蛇添足**了。写出这样语句的人没有很好的领会 IN 的意图。其实 **IN 只关心集合中有没有值，而不关心有几个**。所以此时的 DISTINCT 完全没有必要。那么加上 DISTINCT 会不会对性能产生影响呢？答案是不会。因为 DB2 的优化器会改写上面的 SQL。尽管如此，我们还是不要加 DISTINCT 为好。

特别感谢 CSDN 网友 jackyren007 发现了本文一个非常严重的错误。此错误在 2009-11-19 已经修正。

尽量避免在 SQL 语句中使用 OR

在 SQL 语句中应该尽量避免使用 OR，因为这样做会影响 SQL 语句的性能。考虑下面的情况：

```
1. CREATE TABLE USER
2. (
3. NAME VARCHAR(20) NOT NULL,---姓名
4. BIRTHDAY DATE---生日
5. );
```

现在有这样一个问题：让你查找一下生日是 1949-10-1（共和国同龄人）**或** 1978-12-18（十一届三中全会召开时间）的人，怎么办？
我们很自然就会把这句话翻译成如下 SQL 语句：

```
1. SELECT * FROM USER WHERE BIRTHDAY='1949-10-1' OR BIRTHDAY='1978-12-18'
```

这样做完全正确，可是性能不好，你的思想被这个问题束缚了，我们还可以这么写：

```
1. SELECT * FROM USER WHERE BIRTHDAY IN ('1949-10-1','1978-12-18');
```

有时候，我们不要把自己束缚在问题里面。

尽量避免在 SQL 语句的 WHERE 子句中使用函数

在 SQL 语句的 **WHERE** 子句中应该尽量避免在字段上使用函数，因为这样做会使该字段上的索引失效，影响 SQL 语句的性能。即使该字段上没有索引，也应该避免在字段上使用函数。考虑下面的情况：

```
1. CREATE TABLE USER
2. (
3. NAME VARCHAR(20) NOT NULL,---姓名
4. REGISTERDATE TIMESTAMP---注册时间
5. );
```

现在要求你把 2009.9.24 注册的用户都查出来，怎么办？可能有人会这么写：

```
1. SELECT * FROM USER WHERE REGISTERDATE='2009-9-24';
```

不过很遗憾，这个语句是错误的，因为 REGISTERDATE 是 TIMESTAMP 类型，而'2009-9-24'默认是 DATE 类型，类型不匹配。即然类型不匹配，很自然会想到利用函数进行类型转换，因此，很自然会写出下面的语句：

```
1. SELECT * FROM USER WHERE DATE(REGISTERDATE)='2009-9-24';
```

上述语句完全正确，但是假如 REGISTERDATE 字段上有索引，那么会使索引失效，即使没有索引，也不应该这么做。那么到底如何处理呢？答案是将它转化为范围扫描，如下：

```
1. SELECT * FROM USER WHERE REGISTERDATE>='2009-9-24 00:00:00.0' AND REGI
   STERDATE<'2009-9-25 00:00:00.0';
```

尽量避免在 SQL 语句中使用 LIKE

前面，我们介绍了 尽量避免在 SQL 语句的 WHERE 子句中使用函数，因为这样做会使该字段上的索引失效，影响 SQL 语句的性能。基于同样的道理，我们也应该避免使用 **LIKE**。考虑下面的情况：

```
1. CREATE TABLE USER
2. (
3. NAME VARCHAR(20) NOT NULL,---姓名
4. MYNUMBER VARCHAR(18)---身份证号码
5. );
```

现在要求你把身份证号码开头是 2102（大连人）查出来，怎么办？我们很自然的会这么写：

```
1. SELECT * FROM USER WHERE MYNUMBER LIKE '2102%';
```

上述语句完全正确，只可惜性能不好，那么到底如何处理呢？答案是将它转化为范围扫描，如下：

```
1. SELECT * FROM USER WHERE MYNUMBER>='210200000000000000' AND MYNUMBER<'
   210300000000000000';
```

指定隔离级别

大家应该都知道**隔离级别**吧，至少也应该听说过吧。DB2 共有**四种**隔离级别，**由高到低**分别是：

可重复读（RR）

读稳定性（RS）

游标稳定性（CS）---默认的隔离级别

未落实的读（UR）

隔离级别越低，并发性越好，但是导致的并发性问题也越多。DB2 可以在 **Session（会话级）**、**Connection（连接级）**、**Statement（语句级）** 设定隔离级别。本文只介绍如何指定 **Statement（语句级）** 隔离级别。一般，我们执行 SELECT 语句的时候，为了获得最大的并发性，我们能容忍一些数据不一致，这时我们可以指定最低的隔离级别，如下所示：

```
1.  SELECT * FROM <TABLE-NAME> WITH UR;
```

表连接(JOIN)

了解表连接的人，大概都觉得它很简单。其实简单只是它的外表，如果没有深刻理解 Join 语句，稍不留神就会有逻辑错误，逻辑错误比语法错误更难发现。

要想正确使用 Join 语句，有 2 个知识点是必须掌握的。

第一：SQL 语句执行的顺序

```
1. FROM
2. JOIN ON
3. WHERE
4. GROUP BY
5. HAVING
6. SELECT
7. ORDER BY
8. FETCH FIRST
```

第二：ON 和 WHERE 的区别：ON 是用来定义连接条件的，WHERE 用来过滤结果集

我们来看一个例子，请看如下表定义：

```
1. CREATE TABLE EMPLOY
2. (
3.     NAME    VARCHAR(10),--姓名
4.     DEPTNO    INTEGER--部门编号
5. );
6.
7. INSERT INTO EMPLOY (NAME, DEPTNO) VALUES
8. ('张三',10),
9. ('李四',20),
10.('王五',10),
11.('赵红',20);
12.
13.CREATE TABLE DEPARTMENT
14.(
15.    DEPTNO    INTEGER,--部门编号
16.    DEPTNAME    VARCHAR(10)--部门名
17.);
18.
19.INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME) VALUES
20.(10, '市场部'),
```

```
21. (20, '技术部');
```

我们看一条查询语句：

```
1. SELECT * FROM EMPLOY  E LEFT JOIN DEPARTMENT D
2. ON E.DEPTNO=D.DEPTNO AND D.DEPTNO=40
```

如果你认为上面这条 SQL 语句不会返回任何结果的话，说明你还不清楚 ON 和 WHERE 的区别。**再次声明：ON 是用来定义连接条件的，而不会过滤结果集。**

我们再看一个例子，假设现在让你查询一下所有的员工的姓名和部门名为市场部的部门，怎么查？如下：

```
1. SELECT E.NAME,D.DEPTNAME FROM EMPLOY  E LEFT JOIN DEPARTMENT D ON E.DE
   PTNO=D.DEPTNO
2. WHERE D.DEPTNAME='市场部'
```

如果你也写出上面的语句，那么你就错了，正确的写法应该是这样：

```
1. 方法 1：
2. SELECT E.NAME,D.DEPTNAME FROM EMPLOY  E LEFT JOIN DEPARTMENT D ON E.DE
   PTNO=D.DEPTNO AND D.DEPTNAME='市场部'
3.
4. 方法 2：
5. SELECT E.NAME,D.DEPTNAME FROM EMPLOY  E LEFT JOIN
6. (
7.     SELECT * FROM DEPARTMENT WHERE DEPTNAME='市场部'
8. ) AS D
9. ON E.DEPTNO=D.DEPTNO
```

以上只是举了一个简单的例子，其实，使用 JOIN 的时候，还有好多这样的陷阱，如：在同一个语句中使用 Inner Join 和 Outer Join 的时候，应该特别注意。再如：多个表连接的时候，特别要注意连接条件，如：假设 A、B、C 三个表都有 NO 字段，以下 2 个语句可能产生截然不同的结果。

```
1. 语句 1：
2. A LEFT JOIN B ON A.NO=B.NO
3. LEFT JOIN C ON A.NO=C.NO
4.
5. 语句 2：
6. A LEFT JOIN B ON A.NO=B.NO
7. LEFT JOIN C ON B.NO=C.NO
```

综上所述，使用 Join 的时候特别容易出错，所以我们鼓励，如果需要多表连接时，尽量不使用 JOIN，而使用将连接条件定义在 WHERE 子句中的查询，如：**SELECT * FROM A,B WHERE A.NO=B.NO**。但是，使用这种方式连接表时，也不是没有缺点，它很容易产生笛卡儿乘积，从而使结果集倍增，既使你指定了正确的连接条件，如果连接条件不唯一，也会产生局部笛卡儿乘积，如果这时使用分组统计，很可能产生不正确的结果，所以，也必须加倍小心。

DB2 函数概览

DB2 内置的函数真的是很多，要精通每个函数几乎是不可能的，所以本文并不打算介绍每个函数的具体用法，而是提供一个概览，让您了解每个函数的功能，这样，当你感觉你需要某些功能的函数时，再学习它们的具体用法也不迟。

DB2 内置函数大体分为以下几类：

1. **聚合函数**
2. **类型转换函数**
3. **数学函数**
4. **字符串函数**
5. **日期时间函数**
6. **XML 函数**
7. **分区函数**
8. **安全函数**
9. **其他**

下面我们就了解一下每类都有哪些函数，以及这些函数的功能。

一：聚合函数

二：类型转换函数

DB2 为每种数据类型都提供了相应的函数，一般情况下它们之间的相互转换是非常简单的，请看下表：

```
1. 函数   功能描述
2. SMALLINT     返回 SMALLINT 类型的值
3. INTEGER 返回 INTEGER 类型的值
4. BIGINT   返回 BIGINT 类型的值
5. DECIMAL 返回 DECIMAL 类型的值
6. REAL     返回 REAL 类型的值
7. DOUBLE   返回 DOUBLE 类型的值
8. FLOAT    返回 FLOAT 类型的值
9. CHAR     返回 CHARACTER 类型的值
10. VARCHAR 返回 VARCHAR 类型的值
11. VARCHAR_FORMAT_BIT   将位字符序列格式化为 VARCHAR 类型返回
12. VARCHAR_BIT_FORMAT   将格式化后位字符序列返回到格式化前
```

```
13. LONG_VARCHAR     返回 LONG VARCHAR 类型的值
14. CLOB     返回 CLOB 类型的值
15. GRAPHIC 返回 GRAPHIC 类型的值
16. VARGRAPHIC  返回 VARGRAPHIC 类型的值
17. LONG_VARGRAPHIC 返回 LONG VARGRAPHIC 类型的值
18. DBCLOB  返回 DBCLOB 类型的值
19. BLOB     返回 BLOB 类型的值
20. DATE     返回 DATE 类型的值
21. TIME     返回 TIME 类型的值
22. TIMESTAMP    返回 TIMESTAMP 类型的值
```

三：数学函数

```
1.  函数   功能描述
2.  ABS,ABSVAL  返回参数的绝对值
3.  SIGN     如果参数大于 0 则返回 1，小于 0 返回-1，等于 0 返回 0
4.  RAND     返回 0 和 1 之间的随机浮点数
5.  MOD 求余数
6.  ROUND    返回参数 1 小数点右边的第参数 2 位置处开始的四舍五入值
7.  TRUNCATE OR TRUNC    从表达式小数点右边的位置开始截断并返回该数值
8.  FLOOR    返回小于或等于参数的最大整数
9.  CEILING OR CEIL 返回大于或等于参数的最小的整数值
10. POWER    返回参数 1 的参数 2 次幂
11. SQRT     返回该参数的平方根
12. DIGITS   返回参数绝对值的字符串表示
13. MULTIPLY_ALT     返回参数的乘积
14. DEGREES 求角度
15. RADIANS 将度转换为弧度
16. SIN 正弦函数
17. SINH     双曲线正弦函数
18. ASIN     反正弦函数
19. COS 余弦函数
20. COSH     双曲线余弦函数
21. ACOS     反余弦函数
22. TAN 正切函数
23. TANH     双曲线正切函数
24. ATAN     反正切函数
25. ATANH    双曲线反正切函数
26. ATAN2    反正切函数
27. COT 余切函数
28. LN   返回参数的自然对数
29. LOG 返回参数的自然对数
30. LOG10    返回基于 10 的自然对数
```

31. EXP 返回参数的指数函数

四：字符串函数

1. 函数 功能描述
2. ASCII 将字符转化为 ASCII 码
3. CHR 将 ASCII 码转化为字符
4. STRIP 删除字符串开始和结尾的空白字符或其他指定的字符
5. TRIM 删除字符串开始和结尾的空白字符或其他指定的字符
6. LTRIM 删除字符串开始的空白字符
7. RTRIM 删除字符串尾部的空白字符
8. LCASE or LOWER 返回字符串的小写
9. UCASE OR UPPER 返回字符串的大写
10. SUBSTR 返回子串
11. SUBSTRING 返回子串
12. LEFT 返回开始的 N 个字符
13. RIGHT 返回结尾的 N 个字符
14. POSITION 返回参数 2 在参数 1 中的第一次出现的位置
15. POSSTR 返回参数 2 在参数 1 中的第一次出现的位置
16. LOCATE 返回参数 2 在参数 1 中的第一次出现的位置
17. SPACE 返回由参数指定的长度,包含空格在内的字符串
18. REPEAT 回参数 1 重复参数 2 次后的字符串
19. CONCAT 连接两个字符串
20. INSERT 向指定字符串添加字符串
21. REPLACE 替换字符串
22. TRANSLATE 将字符串中的一个或多个字符替换为其他字符
23. CHARACTER_LENGTH 返回字符串的长度
24. OCTET_LENGTH 返回字符串的字节数
25. ENCRYPT 对字符串加密
26. DECRYPT_BIN and DECRYPT_CHARs 对加密后的数据解密
27. GETHINT 返回密码提示
28. GENERATE_UNIQUE 生成唯一字符序列

五：日期时间函数

1. 函数 功能描述
2. YEAR 返回日期的年部分
3. MONTH 返回日期的月部分
4. DAY 返回日期的日部分
5. HOUR 返回日期的小时部分
6. MINUTE 返回日期的分钟部分

```
7.  SECOND   返回日期的秒部分
8.  MICROSECOND 返回日期的微秒部分
9.  MONTHNAME    返回日期的月份名称
10. DAYNAME 返回日期的星期名称
11. QUARTER 返回指定日期是第几季度
12. WEEK     返回当前日期是一年的第几周，每周从星期日开始
13. WEEK_ISO     返回当前日期是一年的第几周，每周从星期一开始
14. DAYOFWEEK    返回当前日期是一周的第几天，星期日是 1
15. DAYOFWEEK_ISO    返回当前日期是一周的第几天，星期一是 1
16. DAYOFYEAR    返回当前日期是一年的第几天
17. DAYS     返回用整数表示的时间，用来求时间间隔
18. JULIAN_DAY   返回从 January 1, 4712 B.C(Julian date calendar)到指定日期的天
    数
19. MIDNIGHT_SECONDS     返回午夜到指定时间的秒数
20. TIMESTAMPDIFF    返回两个 timestamp 型日期的时间间隔
21. TIMESTAMP_ISO    返回 timestamp 类型的日期
22. TO_CHAR 返回日期的字符串表示
23. VARCHAR_FORMAT   将日期格式化为字符串
24. TO_DATE 将字符串转化为日期
25. TIMESTAMP_FORMAT     将字符串格式化为日期
```

六：XML 函数

七：分区函数

```
1.  函数   功能描述
2.  DATAPARTITIONNUM     返回数据分区中的序列号
3.  DBPARTITIONNUM   返回行的分区号
4.  HASHEDVALUE 返回行的 distribution map index (0 to 4095)
```

八：安全函数

```
1.  函数   功能描述
2.  SECLABEL     返回未命名的安全标签
3.  SECLABEL_BY_NAME     返回具体的安全标签
4.  SECLABEL_TO_CHAR     返回标签的所有元素
```

九：其他

```
1.  函数   功能描述
2.  COALESCE     将 null 转化为其他值
3.  VALUE    将 null 转化为其他值
```

4. NULLIF 如果两个参数相等，则返回 **null**，否则，返回第一个参数
5. HEX 返回一个值的 16 进制表示
6. LENGTH 返回一个值的长度
7. TABLE_NAME 返回 table 名
8. TABLE_SCHEMA 返回 schema 名
9. TYPE_ID 返回数据类型表示
10. TYPE_NAME 返回数据类型名
11. TYPE_SCHEMA 返回 schema 名
12. DEREF 返回参数类型的实例
13. IDENTITY_VAL_LOCAL 返回最后分配给标识列的值
14. REC2XML 返回 XML 标记格式的字符串，包含列名和列数据
15. EVENT_MON_STATE 返回某事件监视器的操作状态
16. RAISE_ERROR 抛出错误，可以指定 sqlstate 和 error_message,有点像 java 的抛出异常

半角全角转换

DB2 的字符串数据类型有单字节和双字节之分。所以，当我们想把**半角变成全角**的时候，其实就是单字节变成双字节，运用我们之前学过的类型转换函数即可。有以下几个函数可用：

```
1. GRAPHIC
2. VARGRAPHIC
3. LONG_VARGRAPHIC
4. DBCLOB
5. 例如：
6. 数字：VALUES VARGRAPHIC('1234567890');
7. 空格：VALUES VARGRAPHIC('　　');
8. 字母：VALUES VARGRAPHIC('abcABC');
9. 日文假名：VALUES VARGRAPHIC('ｶﾅｶﾅ');
```

那么，当我们想把**全角变成半角**的时候，其实就是把双字节变成单字节，运用我们之前学习过的类型转换函数，如下：

```
1. CHAR
2. VARCHAR
3. LONG_VARCHAR
4. CLOB
5. 例如：VALUES VARCHAR('１２３４５６７８９０　ＡＢＣＡＢＣｶﾅｶﾅ');
```

然而，结果却并没有转化成半角，原因是用单引号括起来的字符串，在默认的情况下就是单字节字符串。我们可以这么写：

```
1. VALUES VARCHAR(VARGRAPHIC('1234567890 abcABCｶﾅｶﾅ')) ;
```

不过,遗憾的是以上的**全角转半角**函数只支持 Unicode 编码的数据库。

将 null 值转化为其他值

我个人认为数据库中不应该有 null 值，因为他颠覆了二值逻辑结构（即：真和假），出现了三值逻辑结构（即：真、假和未知）。由于 null，我们的 SQL 语句很有可能出现意想不到的结果。此外 null 值和其他值进行数值运算的时候也会带来问题。但是，有时候有些事情并不是我们能够控制和改变的，作为一名真正的 SQL 开发者，应该敢于面对最垃圾的数据库设计。下面给大家介绍如何将 null 值转化为其他值。
有两个函数可以将 null 值转化为其它值：**VALUE、COALESCE,**它们的**使用方法**和**作用**完全相同。

```
1. 语法：coalesce(<field>，<target_value>)
2. 语法：value(<field>，<target_value>)
3. 例如：select coalesce(id,0) from <table_name>
4. 例如：select value(id,0) from <table_name>
5. 说明：如果 id 为 null，则把它转化为 0，否则的话不变。
```

考虑下面的情况：

```
1. CREATE TABLE USER
2. (
3. NAME VARCHAR(20) NOT NULL,---姓名
4. SALARY1 FLOAT,---基本工资
5. SALARY2 FLOAT---奖金
6. );
```

假设你要查找总工资(基本工资+奖金)大于 3000 元的员工，我们很自然的会写出下面的语句：

```
1. SELECT NAME FROM USER WHERE SALARY1+SALARY2>3000;
```

但是很不幸，这条语句并不是永远正确，当 SALARY1 或 SALARY2 有一个值是 null 的时候，我们很可能会漏掉部分数据（会漏掉哪些数据呢？朋友们自己思考一下），这就是我认为数据库中不应该有 null 值的原因，如果你不是决策者，无法改变数据库设计，我们可以这样写：

```
1. SELECT NAME FROM USER WHERE COALESCE(SALARY1,0)+COALESCE(SALARY2,0)>30
   00;
2. 或
```

```
3. SELECT NAME FROM USER WHERE VALUE(SALARY1,0)+COALESCE(SALARY2,0)>3000;
```

这样就可以保证查询出的结果就是我们想要的结果。

操作日期和时间

我们都知道数字可以进行加、减、乘、除等运算。那么，日期可不可以呢？答案是，日期只能进行加、减运算。

在开始操作日期之前，我们得先了解 DB2 支持哪些日期数据类型，如下所示：

类型	格式
TIME	hh:mm:ss
DATE	yyyy-mm-dd
TIMESTAMP	yyyy-mm-dd hh:mm:ss.zzzzzz

下面，我们还是先看一个简单的例子吧，如下所示：

```
1. VALUES DATE('2009-10-1') + 1 DAY
2. VALUES DATE('2009-10-1') + 5 DAYS
```

如上所示，我们可以以人类英语的方式来操作日期，那么，除了 DAY 和 DAYS 之外，DB2 还支持哪些关键字呢？如下所示：

单数	复数
YEAR	YEARS
MONTH	MONTHS
DAY	DAYS
HOUR	HOURS
MINUTE	MINUTES
SECOND	SECONDS
MICROSECOND	MICROSECONDS

单数和复数关键字在使用上没有任何区别，之所以分单数和复数，可能是考虑英语的语言习惯。下面是一些简单的例子：

```
1. VALUES DATE('2009-10-1') + 1 YEAR + 2 MONTH -8 DAY
2. VALUES TIME('10:23:15') + 3 HOUR -26 MINUTE
3. VALUES TIMESTAMP('2009-10-1 10:23:15.000000') - 3 SECOND + 450 MICROSE
   COND
```

怎么样？使用起来是不是非常简单。不过，还有个问题我们需要思考，如果给 DATE 类型日期加上 2 小时，或者给 TIME 类型的日期加上 1 年 会出现怎样的情况呢？请读者自己运行下面的代码找答案吧。

```
1. VALUES DATE('2009-10-1') + 2 HOUR
2. VALUES TIME('10:23:15') + 1 YEAR
```

有时候，我们需要知道两个日期之间相隔多少天，也就是说求日期之间的时间间隔，该怎么办呢？我们很自然的想到把两个日期相减，如下所示：

```
1. VALUES DATE('2009-10-1') -DATE('2008-10-1') ;
2. VALUES TIME('10:23:15')  - TIME('10:22:15') ;
3. VALUES TIMESTAMP('2009-10-1 10:23:15.000005')  - TIMESTAMP('2009-10-1
    10:22:15.000000') ;
```

运行上面的语句后，你可能感觉很失望，答案并不是我们期望的结果。上面的语句，首先将日期类型转换为 DECIMAL 类型，然后进行减法运算。转换规则如下：

日期	DECIMAL	转换后的格式
DATE	DECIMAL(8,0)	yyyymmdd
TIME	DECIMAL(6,0)	hhmmss
TIMESTAMP	DECIMAL(20,6)	yyyymmddhhmmss.zzzzzz

所以，上面的三条语句的结果如下：

```
1. 20091001-20081001=10000;
2. 102315-102215=100;
3. 20091001102315.000005-20091001102215.000000=100.000005;
```

那么，到底该怎样求时间间隔呢？下面给大家介绍两个函数：**DAYS 和 TIMESTAMP DIFF**

DAYS 函数可以用来求两个日期的天数，如下 SQL 所示：

```
1. VALUES DAYS(DATE('2009-10-1') ) - DAYS(DATE('2008-10-1')) ;
2. VALUES DAYS('2009-10-2' ) - DAYS('2009-10-1 10:40:15.000000') ;
3. VALUES DAYS(TIMESTAMP('2009-10-2 10:23:15.000005'))  - DAYS(TIMESTAMP
   ('2009-10-1 10:40:15.000000'));
```

DAYS 函数只能用来求两个日期之间天数，不够灵活。更灵活的是 TIMESTAMPDIFF 函数，TIMESTAMPDIFF 函数的定义如下：

TIMESTAMPDIFF（参数 1，参数 2）

参数 1 可以指定为：1、2、4、8、16、32、64、128、256，分别表示返回两个日期之间的 毫秒数、秒数、分钟数、小时数、天数、周数、月数、季度数、年数。

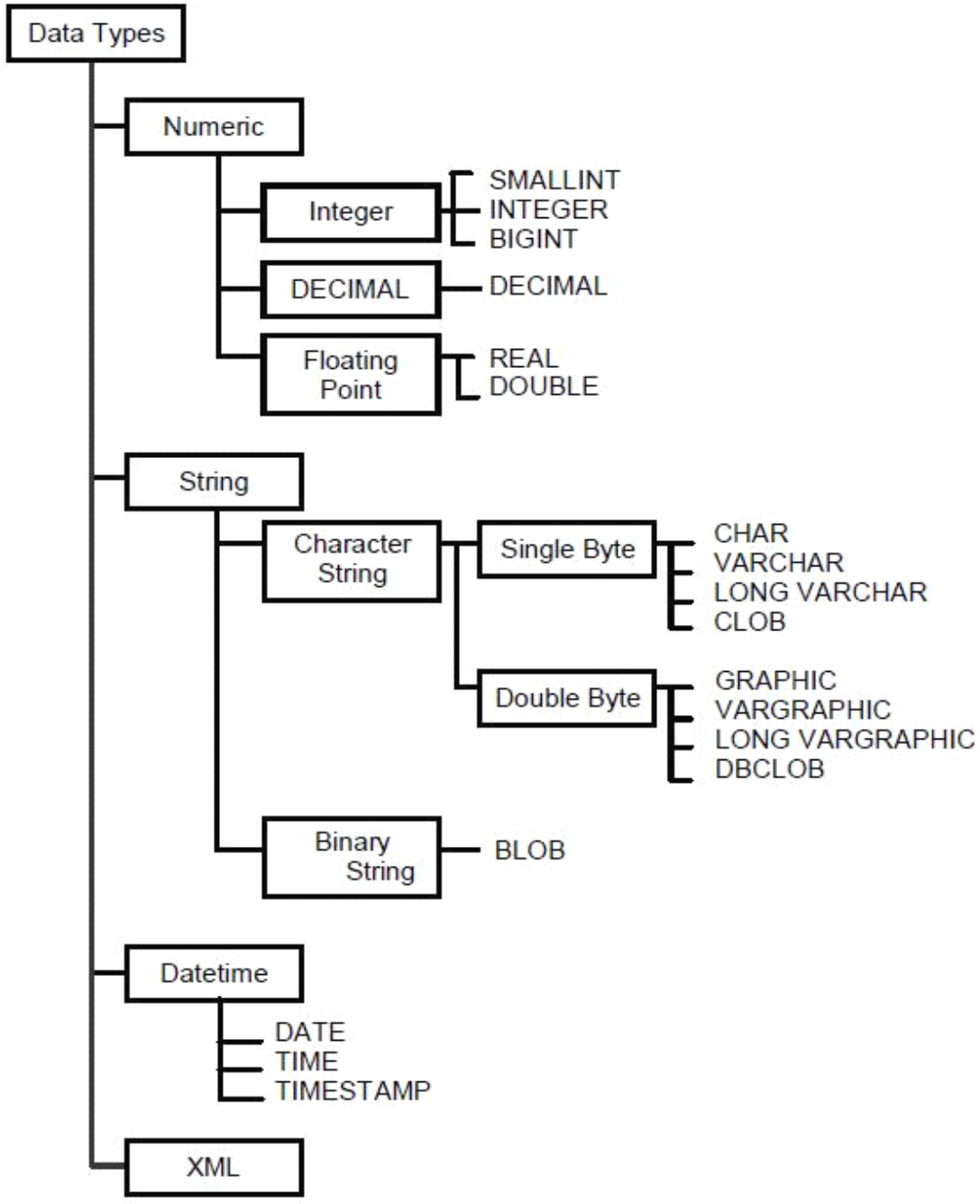
参数 2 是两个日期相减的结果，如下 SQL 所示：

```
1. VALUES TIMESTAMPDIFF(1, CHAR(TIMESTAMP('2009-10-1 10:23:16.000000') -
   TIMESTAMP('2009-10-1 10:23:15.000000')));
2. VALUES TIMESTAMPDIFF(2, CHAR(TIMESTAMP('2009-10-1 10:23:15.000000') -
   TIMESTAMP('2008-10-1 10:23:15.000000')));
3. VALUES TIMESTAMPDIFF(4, CHAR(TIMESTAMP('2009-10-1 10:23:15.000000') -
   TIMESTAMP('2008-10-1 10:23:15.000000')));
4. VALUES TIMESTAMPDIFF(8, CHAR(TIMESTAMP('2009-10-1 10:23:15.000000') -
   TIMESTAMP('2008-10-1 10:23:15.000000')));
5. VALUES TIMESTAMPDIFF(16, CHAR(TIMESTAMP('2009-10-1 10:23:15.000000') -
   TIMESTAMP('2008-10-1 10:23:15.000000')));
6. VALUES TIMESTAMPDIFF(32, CHAR(TIMESTAMP('2009-10-1 10:23:15.000000') -
   TIMESTAMP('2008-10-1 10:23:15.000000')));
7. VALUES TIMESTAMPDIFF(64, CHAR(TIMESTAMP('2009-10-1 10:23:15.000000') -
   TIMESTAMP('2008-10-1 10:23:15.000000')));
8. VALUES TIMESTAMPDIFF(128, CHAR(TIMESTAMP('2009-10-1 10:23:15.000000')
    -TIMESTAMP('2008-10-1 10:23:15.000000')));
9. VALUES TIMESTAMPDIFF(256, CHAR(TIMESTAMP('2009-10-1 10:23:15.000000')
    -TIMESTAMP('2008-10-1 10:23:15.000000')));
```

有关日期的操作还有很多，请参考日期时间函数。

数据类型转换

说起数据类型转换，我们得先了解一下 DB2 有哪些数据类型，请看下图：



DB2 内置数据类型

从上图我们可以看出，DB2 有四种数据类型，分别为**数字、字符、日期、XML**，由于 XML 数据类型很少使用，暂时我们不讨论，本文主要讨论**数字、字符、日期** 之间的相互转换。

DB2 为每种数据类型都提供了相应的函数，一般情况下它们之间的相互转换是非常简单的，DB2 提供的函数有：

```
1. SMALLINT
2. INTEGER
3. BIGINT
4. DECIMAL
5. REAL
6. DOUBLE
7. CHAR
8. VARCHAR
9. LONG_VARCHAR
10. CLOB
11. GRAPHIC
12. VARGRAPHIC
13. LONG_VARGRAPHIC
14. DBCLOB
15. BLOB
16. DATE
17. TIME
18. TIMESTAMP
```

下面，我们看个简单的例子：

```
1. SELECT * FROM
2. (
3. --序号     姓名     生日
4. VALUES
5. ('11','WAVE','1997-7-1'),
6. ('2','WAVE','1949-10-1')
7. ) AS USER(ID,NAME,BIRTHDAY)
8. ORDER BY USER.ID;
```

运行上面的 SQL 后，我们发现，它的排序方式并不是我们想要的排序方式，原因是**序号（ID）**列为字符型，要想得到我们想要的排序方式，我们可以将**序号（ID）**转化为**数值型，**如下：

```
1. SELECT * FROM
2. (
3. --序号     姓名     生日
4. VALUES
5. ('11','WAVE','1997-7-1'),
6. ('2','WAVE','1949-10-1')
7. ) AS USER(ID,NAME,BIRTHDAY)
```

```
8. ORDER BY INTEGER(USER.ID);
```

运行上面的 SQL 后，我们发现正是我们期望的结果。上面这些函数使用起来是非常简单的，这里我们就不一一介绍了。下面我们学习数据类型转换的另一种方法，CAST 表达式，如下 SQL：

```
1. SELECT
2.     ID,
3.     NAME AS NAME1,
4.     CAST(NAME AS CHAR(4) )AS NAME2,
5.     SALARY AS SALARY1,
6.     CAST(SALARY AS INTEGER) AS SALARY2,
7.     CAST(NULL AS INTEGER) AS TEST_NULL
8. FROM
9. (
10.    --员工号 姓名 工资
11.    VALUES
12.    ('11','ZHANGSAN',3000.5),
13.    ('2','WANGWU',4000.1)
14.) AS USER (ID,NAME,SALARY)
15.ORDER BY CAST(ID AS INTEGER);
```

千万别着急运行上面的 SQL，运行之前请仔细分析和思考一下这条语句的结果，然后再运行，看看和你分析的有什么不同。
上面的语句有四处用到了 CAST 表达式，如下：

```
1. CAST(NAME AS CHAR(4) )AS NAME2---将 VARCHAR 转换为 CHAR
2. CAST(SALARY AS INTEGER) AS SALARY2, ---将 DOUBLE 转换为 INTEGER
3. CAST(NULL AS INTEGER) AS TEST_NULL---将 NULL 转换为 INTEGER
4. ORDER BY CAST(ID AS INTEGER); ---将 VARCHAR 转换为 INTEGER
```

分析结果我们发现，某些情况会发生数据截断，所以，使用 CAST 的时候请千万注意。

SQL 中的 IF ELSE（CASE 语句的使用）

大家对 IF ELSE 语句可能都很熟悉，它是用来对过程进行控制的。在 SQL 的世界中 CASE 语句语句有类似的效果。下面简单的介绍 CASE 语句的用法。考虑下面的情况，假设有个 user 表，定义如下：

```
1. CREATE TABLE USER
2. (
3. NAME VARCHAR(20) NOT NULL,---姓名
4. SEX INTEGER,---性别（1、男    2、女）
5. BIRTHDAY DATE---生日
6. );
```

CASE 使用场合 1：把 user 表导出生成一个文件，要求性别为男或女，而不是 1 和 2，怎么办？我们可以用如下的语句处理：

```
1. SELECT
2.     NAME,
3.     CASE SEX
4.         WHEN 1 THEN '男'
5.         ELSE '女'
6.     END AS SEX,
7.     BIRTHDAY
8. FROM USER;
```

CASE 使用场合 2：假设 user 目前没有值，然后你往 user 导入了一批数据，但是很不幸，错把男设置成为 2，而把女设置成为 1，现在要求你变换过来，怎么办？
方法 1：使用三条语句，先把 2 更新成 3，接着把 1 更新成 2，最后把 3 更新成 1，很麻烦，不是吗？

```
1. UPDATE USER SET SEX=3 WHERE SEX=2;
2. UPDATE USER SET SEX=1 WHERE SEX=3;
3. UPDATE USER SET SEX=2 WHERE SEX=1;
```

方法 2：使用 CASE 语句

```
1. UPDATE USER SET SEX=
2. (
3. CASE SEX
4. WHEN 1 THEN 2
5. WHEN 2 THEN 1
6. ELSE SEX
7. END
8. );
```

细心的朋友可能已经发现了，上面的方法 1 的三条语句的执行顺序有问题，没错，是我故意那些写的，仅仅是把 1 变成 2，把 2 变成 1 就那么麻烦，而且很容易出错，想象一下，如果有很多这样的值需要变换，那是一种什么样的情况。还好，我们有 CASE 语句，有好多这样的值需要变换，CASE 语句也不会存在问题。可能有些朋友还是有疑虑，这样做会不会死循环啊？哈哈，想法很好，如果你发现这样做会死循环，一定要告诉 IBM，我反正没发现。

CASE 使用场合 3：假设让你把张三的生日更新成 1949-10-1，李四的生日更新成 1997-7-1 等，类似这样的更新由很多。该怎么办呢？非常简单，大多数人会这么做。

```
1. update USER set BIRTHDAY='1949-10-1' where NAME='张三';
2. update USER set BIRTHDAY='1997-7-1' where NAME='李四';
```

当 USER 表的数据量非常大，而 NAME 字段上又没有索引时，每条语句都要进行全表扫描，如果这样的语句有很多，效率会非常差，这时候我们可以用 CASE 语句，如下：

```
1. UPDATE USER SET BIRTHDAY=
2. (
3. CASE NAME
4. WHEN '张三' THEN '1949-10-1'
5. WHEN '李四' THEN '1997-7-1'
6. ELSE BIRTHDAY
7. END
8. )
9. where NAME in ('张三','李四');
```

以上语句只进行一次全表扫描，效率非常高。

感谢 CSDN 网友 higny 发现了本文的一个错误，在此表示严重感谢。

定义临时集合（VALUES 语句的使用）

提起 **VALUES 语句**，很多人都感觉非常陌生，哈哈，看到下面的语句，你就会恍然大悟。

```
1. INSERT INTO USER (NAME,BIRTHDAY) VALUES ('张三','2000-1-1');
```

哦，原来这就是 VALUES 语句啊，没错，这就传说中的 VALUES 语句，那它有什么用途呢？正如我们文章的标题，它是用来定义临时集合的。
我们先来看几个简单的 VALUES 语句，如下：

```
1. VALUES 1                        --1 行 1 列
2. VALUES 1, 2                  --1 行 2 列
3. VALUES (1), (2)           --2 行 1 列
4. VALUES (1,2), (1,3),(2,1) --3 行 2 列
```

是不是有点乱啊，那我们缕一缕，把它当普通 SQL 一样执行一下，是不是很直观（**不要以为它只能看，可以执行的**）。那大家先猜一猜以下语句是几行几列呢？

```
1. VALUES ((1), (2))
2. VALUES ((1,2), (1,3),(2,1))
```

执行一下，看和你想的一样不一样啊。之前我们说过，VALUES 语句定义的是临时集合，而我们知道集合是可以排序、分组的，那 VALUES 语句可不可以呢？你可以试一试如下语句：

```
1. ---排序
2. SELECT * FROM
3. (
4. VALUES (1,2), (2,1)
5. ) AS TEMP
6. ORDER BY 1 DESC
7. ---分组
8. SELECT A,COUNT(*) FROM
9. (
10.VALUES (1,2), (1,3),(2,1)
11.) AS TEMP(A,B)
12.GROUP BY A
```

看到这里你应该学会了定义 VALUES 语句，你可能还想知道，在实际环境中，哪些情况下我们该使用 VALUES 语句呢？答案可能令你失望，答案就是任何需要临时表的时候都可以。举个简单的例子，考虑下面的情况：

```
CREATE TABLE USER
(
NAME VARCHAR(20) NOT NULL,---姓名
DEPARTMENT INTEGER,---部门（1、市场部    2、管理部    3、研发部）
BIRTHDAY DATE---生日
);
```

现在给你以下条件，让你把姓名查出来：
部门　　生日
市场部　1949-10-1
管理部　　1978-12-18
研发部　　1997-7-1
… …
类似这样的条件有很多，我们就以上面的三个条件举例。该怎么办呢？有些人可能会这么写：

```
SELECT * FROM USER WHERE DEPARTMENT IN (1,2,3) AND BIRTHDAY IN ('1949-10-1','1978-12-18','1997-7-1');
```

查询出来后发现结果不正确，因为把管理部，生日是 1949-10-1 也查出来了。既然这么处理不行，有人可能会这么写：

```
SELECT * FROM USER WHERE (DEPARTMENT,BIRTHDAY) IN
(
(1,'1949-10-1'),
(2,'1978-12-18'),
(3,'1997-7-1')
);
```

结果发现这条语句根本就不能执行，有人可能会说没办法了，一条一条执行吧，如下这样：

```
SELECT * FROM USER WHERE DEPARTMENT=1 and BIRTHDAY='1949-10-1';
SELECT * FROM USER WHERE DEPARTMENT=2 and BIRTHDAY='1978-12-18';
SELECT * FROM USER WHERE DEPARTMENT=3 and BIRTHDAY='1997-7-1';
```

经过漫长的等待（因为这样效率很差），终于查出来了，结果发现怎么还有些我们不想要的内容，如：换行，甚至是 DB2 打印出的一些消息。基于以上缺点，聪明人想出一个好方法，新建一个表（如：temp），把以上条件导入，然后在查询，不就可以了吗？如下：

```
CREATE TABLE TEMP
(
DEPARTMENT INTEGER,
```

```
4. BIRTHDAY DATE
5. );
```

然后把条件导入到临时表中，最后这样查询：

```
1. SELECT * FROM USER WHERE (DEPARTMENT,BIRTHDAY) IN
2. (
3. SELECT DEPARTMENT,BIRTHDAY FROM TEMP
4. );
```

除了麻烦点，一切似乎很完美。不过，知道 VALUES 语句的人会说：这样做太麻烦，不用定义持久表，用 VALUES 定义一个临时的集合不就可以了，如下：

```
1. SELECT * FROM USER WHERE (DEPARTMENT,BIRTHDAY) IN
2. (
3. VALUES (1,'1949-10-1'),(2,'1978-12-18'),(3,'1997-7-1')
4. );
```

至此，我们感觉这样做已经很简单了，不过，不一定，还有一种更简单的方法，如下：

```
1. SELECT * FROM USER WHERE DEPARTMENT=1 AND BIRTHDAY='1949-10-1'
2. UNION
3. SELECT * FROM USER WHERE DEPARTMENT=2 AND BIRTHDAY='1978-12-18'
4. UNION
5. SELECT * FROM USER WHERE DEPARTMENT=3 AND BIRTHDAY='1997-7-1'
```

当你看到这的时候，本文也该结束了，你有什么启发呢？

DB2 公共表表达式（WITH 语句的使用）

说起 WITH 语句，除了那些第一次听说 WITH 语句的人，大部分人都觉得它是用来做递归查询的。其实那只是它的一个用途而已，它的本名正如我们标题写的那样，叫做：**公共表表达式（Common Table Expression）**，从字面理解，大家觉得它是用来干嘛的呢？其实，它是用来定义**临时集合**的。啊？**VALUES 语句**不是用来定义临时集合的吗？怎么 WITH 语句也用来定义临时集合呢？它们有什么**区别**呢？

VALUES 语句是用**明确的值**来定义临时集合的，如下：

```
1. VALUES (1,2), (1,3),(2,1)
```

WITH 语句是用**查询**(也就是 select 语句)来定义临时集合的，从这个角度讲，有点像视图，不过不是视图，大家千万别误解。如下：

```
1. CREATE TABLE USER (
2. NAME VARCHAR(20) NOT NULL,---姓名
3. SEX INTEGER,---性别（1、男   2、女）
4. BIRTHDAY DATE---生日
5. );

1. WITH TEST(NAME_TEST, BDAY_TEST) AS
2. (
3. SELECT NAME,BIRTHDAY FROM USER--语句 1
4. )
5. SELECT NAME_TEST FROM TEST WHERE BDAY_TEST='1949-10-1';--语句 2
```

下面我们来解释一下，首先**语句 1** 执行，它会产生一个有两列（NAME,BIRTHDAY）的结果集；接着，我们将这个结果集命名为 test，并且将列名重命名为 NAME_TEST, BDAY_TEST；最后我们执行**语句 2**，从这个临时集合中找到生日是 1949-10-1，也就是共和国的同龄人。

怎么样？如果你感觉不好理解，请仔细的分析一下上面的语句。下面我们举个 **VALUES 语句**和 **WITH 语句**结合使用的例子，如下：

```
1. WITH TEST(NAME_TEST, BDAY_TEST) AS
2. (
3. VALUES ('张三','1997-7-1'),('李四','1949-10-1')
4. )
5. SELECT NAME_TEST FROM TEST WHERE BDAY_TEST='1949-10-1'
```

从上面的介绍和 WITH 语句不为大多数人所熟悉可以猜测，WITH 语句是为复杂的查询为设计的，的确是这样的，下面我们举个复杂的例子，想提高技术的朋友可千万不能错过。考虑下面的情况：

```
1. CREATE TABLE USER
2. (
```

```
3.  NAME VARCHAR(20) NOT NULL,--姓名
4.  DEGREE INTEGER NOT NULL,--学历(1、专科 2、本科 3、硕士 4、博士)
5.  STARTWORKDATE date NOT NULL,--入职时间
6.  SALARY1 FLOAT NOT NULL,--基本工资
7.  SALARY2 FLOAT NOT NULL--奖金
8.  );
```

假设现在让你查询一下那些 1、学历是硕士或博士 2、学历相同，入职年份也相同，但是工资（基本工资+奖金）却比**相同条件员工的平均工资**低的员工。（哈哈，可能是要涨工资），不知道你听明白问题没有？该怎么查询呢？我们是这样想的：

1、查询学历是硕士或博士的那些员工得到**结果集 1**，如下：

```
1.  SELECT NAME,DEGREE,YEAR(STARTWORKDATE) AS WORDDATE, SALARY1+SALARY2 AS
     SALARY FROM USER WHERE DEGREE IN (3,4);
```

2、根据学历和入职年份分组，求平均工资 得到**结果集 2**，如下：

```
1.  SELECT DEGREE,YEAR(STARTWORKDATE) AS WORDDATE, AVG(SALARY1+SALARY2) AS
     AVG_SALARY
2.  FROM USER WHERE DEGREE IN (3,4)
3.  GROUP BY DEGREE,YEAR(STARTWORKDATE)
```

3、以**学历**和**入职年份**为**条件 联合**两个结果集，**查找工资<平均工资** 的员工，以下是完整的 SQL：

```
1.  WITH TEMP1(NAME,DEGREE,WORDDATE,SALARY) AS
2.  (
3.  SELECT NAME,DEGREE,YEAR(STARTWORKDATE) AS WORDDATE, SALARY1+SALARY2 AS
     SALARY FROM USER WHERE DEGREE IN (3,4)
4.  ),
5.  TEMP2 (DEGREE,WORDDATE,AVG_SALARY) AS
6.  (
7.  SELECT DEGREE,YEAR(STARTWORKDATE) AS WORDDATE, AVG(SALARY1+SALARY2) AS
     AVG_SALARY
8.  FROM USER WHERE DEGREE IN (3,4)
9.  GROUP BY DEGREE,YEAR(STARTWORKDATE)
10. )
11. SELECT NAME FROM TEMP1, TEMP2 WHERE
12. TEMP1.DEGREE=TEMP2.DEGREE
13. AND TEMP1.WORDDATE=TEMP2.WORDDATE
14. AND SALARY<AVG_SALARY;
```

查询结果完全正确，但我们还有**改善的空间**，在查询**结果集 2** 的时候，我们是从 user 表中取得数据的。其实此时结果集 1 已经查询出来了，我们完全可以从**结果集 1** 中通过分组得到结果集 2，而不用从 uer 表中得到结果集 2，比较上面和下面的语句你就可以知道我说的是什么意思了！

```
1. WITH TEMP1(NAME,DEGREE,WORDDATE,SALARY) AS
2. (
3. SELECT NAME,DEGREE,YEAR(STARTWORKDATE) AS WORDDATE, SALARY1+SALARY2 AS
    SALARY FROM USER WHERE DEGREE IN (3,4)
4. ),
5. TEMP2 (DEGREE,WORDDATE,AVG_SALARY) AS
6. (
7. SELECT DEGREE,WORDDATE, AVG(SALARY) AS AVG_SALARY
8. FROM TEMP1
9. GROUP BY DEGREE,WORDDATE
10.)
11.SELECT NAME FROM TEMP1, TEMP2 WHERE
12.TEMP1.DEGREE=TEMP2.DEGREE
13.AND TEMP1.WORDDATE=TEMP2.WORDDATE
14.AND SALARY<AVG_SALARY;
```

可能有些朋友会说，我不用 WITH 语句也可以查出来，的确是这样，如下：

```
1. SELECT U.NAME FROM USER AS U,
2. (
3. SELECT DEGREE,YEAR(STARTWORKDATE) AS WORDDATE, AVG(SALARY1+SALARY2) AS
    AVG_SALARY
4. FROM USER WHERE DEGREE IN (3,4)
5. GROUP BY DEGREE,YEAR(STARTWORKDATE)
6. ) AS G
7. WHERE U.DEGREE=G.DEGREE
8. AND YEAR(U.STARTWORKDATE)=G.WORDDATE
9. AND (SALARY1+SALARY2)<G.AVG_SALARY;
```

那使用 WITH 和不使用 WITH，这两种写法有什么区别呢？一般情况下这两种写法在性能上不会有太大差异，但是，

1、当 USER 表的记录很多

2、硕士或博士（DEGREE IN (3,4)）在 USER 表中的比例很少

当满足以上条件时，这两种写法在性能的差异将会显现出来，为什么呢？因为不使用 WITH 写法的语句**访问了 2 次 USER 表**，如果 DEGREE 字段又没有索引，性能差异将会非常明显。

当你看到这时，如果很好的理解了上面的内容，我相信你会对 WITH 语句有了一定的体会。然而 WITH 语句能做的还不止这些，下面给大家介绍一下，如何用 WITH 语句做**递归查询**。递归查询的一个典型的例子是对**树状结构的表**进行查询，考虑如下的情况：

```
论坛首页
--数据库开发
----DB2
------DB2 文章 1
--------DB2 文章 1 的评论 1
--------DB2 文章 1 的评论 2
------DB2 文章 2
----Oracle
--Java 技术
```

以上是一个论坛的典型例子，下面我们新建一个表来存储以上信息。

```
CREATE TABLE BBS
(
PARENTID INTEGER NOT NULL,
ID INTEGER NOT NULL,
NAME VARCHAR(200) NOT NULL---板块、文章、评论等。
);
insert into bbs (PARENTID,ID,NAME) values
(0,0,'论坛首页'),
(0,1,'数据库开发'),
(1,11,'DB2'),
(11,111,'DB2 文章 1'),
(111,1111,'DB2 文章 1 的评论 1'),
(111,1112,'DB2 文章 1 的评论 2'),
(11,112,'DB2 文章 2'),
(1,12,'Oracle'),
(0,2,'Java 技术');
```

现在万事兼备了，我们开始查询吧。假设现在让你**查询一下‘DB2 文章 1’**的所有评论，有人说，这还不简单，如下这样就可以了。

```
SELECT * FROM BBS WHERE PARENTID=(SELECT ID FROM BBS WHERE NAME='DB2
');
```

答案完全正确。那么，现在让你**查询一下 DB2** 的所有文章及评论，怎么办？传统的方法就很难查询了，这时候递归查询就派上用场了，如下：

```
1. WITH TEMP(PARENTID,ID,NAME) AS
2. (
3. SELECT PARENTID,ID,NAME FROM BBS WHERE NAME='DB2'---语句 1
4. UNION ALL---语句 2
5. SELECT B.PARENTID,B.ID,B.NAME FROM BBS AS B, TEMP AS T WHERE B.PARENTI
   D=T.ID---语句 3
6. )
7. SELECT NAME FROM TEMP;---语句 4
```

运行后，我们发现，结果完全正确，那它到底是怎么运行的呢？下面我们详细讲解一下。
1、首先，**语句 1** 将会执行，**它只执行一次**，作为**循环的起点**。得到结果集：DB2
2、接着，将**循环执行语句 3**，这里我们有必要详细介绍一下。
首先**语句 3 的意图**是什么呢？说白了，它就是**查找语句 1 产生结果集（DB2）的下一级**，那么在目录树中 DB2 的下一级是什么呢？是‘DB2 文章 1’和‘DB2 文章 2’，并且把查询到的结果集作为**下一次循环的起点**，然后查询它们的下一级，直到没有下一级为止。
怎么样？还没明白？哈哈，不要紧，我们一步一步来：
首先，**语句 1 产生结果集**：DB2，作为循环的起点，把它和 BBS 表关联来查找它的下一级，查询后的结果为：‘DB2 文章 1’和‘DB2 文章 2’
接着，把**上次的查询结果**（也就是‘DB2 文章 1’和‘DB2 文章 2’）和 BBS 表关联来查找它们的下一级，查询后的结果为：‘DB2 文章 1 的评论 1’ 和 ‘DB2 文章 1 的评论 2’。
然后，在把**上次的查询结果**（也就是‘DB2 文章 1 的评论 1’ 和 ‘DB2 文章 1 的评论 2’）和 BBS 表关联来查找它们的下一级，此时，没有结果返回，**循环结束**。
3、第三，将执行**语句 2**，将所有的结果集放在一起，最终得到 temp 结果集。
4、最后，我们通过**语句 4** 从 temp 临时集合中得到我们期望的查询结果。
怎么样，这回理解了吧，如果还没有理解，那么我也无能为力了。需要特别提醒的是
1、一定要注意**语句 3 的关联条件**，否则很容易就写成**死循环**了。
2、语句 2 必须是 UNION ALL
最后请大家猜想一下，把**语句 1 的 where 子句**去掉，将会产生什么样的结果呢？

嵌套表表达式（**Nested Table Expression**）

```
1. SELECT * FROM <TABLE-NAME>;
```

看到上面的语句了吗？这是我们在熟悉不过的一条语句，我们中的大多人学习 SQL 正是从这条语句开始的。所以大多数人认为 FROM 语句后只能接一个表或视图（或者压根就没多想），有这种想法的人我非常能理解，因为我曾经也是这其中的一员。其实 FROM 后面可以接任何**集合（表）**。说到这，**关于集合和表**，我特别想多少几句。SQL 的理论基础是数学中的集合理论，所以 SQL 关注的就是如何对集合进行操作，基于以上原因，我特别喜欢名词 集合，而不喜欢说表。不过，大家如果不习惯，也可以把集合和表当一个意思来理解，因为我们不是搞理论研究工作的，没必要深究他们之间的细微差别。说了这么多，我们还是赶快来看个例子吧。

```
1. ---建表
2. CREATE TABLE USER
3. (
4. NAME VARCHAR(20) NOT NULL,---姓名
5. BIRTHDAY DATE---生日
6. );
7.
8.
9. ---例子 1
10.SELECT * FROM
11.(
12.SELECT * FROM USER
13.) AS TEMP1;
14.
15.
16.---例子 2
17.SELECT NAME,BIRTHDAY FROM
18.(
19.SELECT NAME,BIRTHDAY FROM USER
20.) AS TEMP1;
21.
22.
23.---例子 3
24.SELECT TEMP1.NAME,TEMP1.BIRTHDAY FROM
25.(
26.SELECT NAME,BIRTHDAY FROM USER
27.) AS TEMP1;
```

```
28.
29.
30. ---例子 4
31. SELECT N,B FROM
32. (
33. SELECT NAME,BIRTHDAY FROM USER
34. ) AS TEMP1(N,B);
35.
36.
37. ---例子 5
38. SELECT AAA.A,AAA.B,XXX.X,XXX.Y FROM
39. (
40. SELECT NAME,BIRTHDAY FROM USER
41. ) AS AAA(A,B),--集合 1
42. (
43. SELECT NAME,BIRTHDAY FROM USER
44. ) AS XXX(X,Y)--集合 2
45. WHERE AAA.A=XXX.X AND AAA.B=XXX.Y--关联两个集合
```

看到上面的例子 5 了吗？我们可以给临时集合命名，还可以给重命名临时集合的列名，还可以关联两个临时集合，总之，操作临时集合和操作表没有区别。

DB2 临时表

临时表（TEMPORARY TABLE）通常应用在需要定义临时集合的场合。但是，在大部分需要临时集合的时候，我们根本就不需要定义临时表。当我们在一条 SQL 语句中只使用一次临时集合时，我们可以使用**嵌套表表达式**来定义临时集合；当我们在一条 SQL 语句中需要多次使用同一临时集合时，我们可以使用**公共表表达式**；只有当我们在一个工作单元中的多条 SQL 语句中使用同一临时集合时，我们才需要定义临时表。

可以通过以下三种方式定义临时表：

```
方法 1：
DECLARE GLOBAL TEMPORARY TABLE SESSION.EMP
(
    NAME VARCHAR(10),---姓名
    DEPT SMALLINT,---部门
    SALARY DEC(7,2)---工资
)
ON COMMIT DELETE ROWS;

方法 2：
DECLARE GLOBAL TEMPORARY TABLE session.emp
LIKE staff INCLUDING COLUMN DEFAULTS
WITH REPLACE
ON COMMIT PRESERVE ROWS;

方法 3：
DECLARE GLOBAL TEMPORARY TABLE session.emp AS
(
    SELECT * FROM staff WHERE <condition>
)
DEFINITION ONLY
WITH REPLACE;
```

定义了临时表后，我们可以像使用普通表一样使用临时表。临时表只对定义它的用户有效，不同用户可以在同一时间定义同名的临时表，他们之间互不影响。临时表的生命周期是 SESSION，当 SESSION 关闭时，临时表将自动删除，这也是临时表的模式名只能为 SESSION 的原因。此外，我们还可以给临时表定义索引。更多细节请参考 DB2 信息中心 。

DB2 在线分析处理（OLAP 函数的使用）

说起 DB2 在线分析处理，可以用很好很强大来形容。这项功能特别适用于各种统计查询，这些查询用通常的 SQL 很难实现，或者根本就无发实现。首先，我们从一个简单的例子开始，来一步一步揭开它神秘的面纱，请看下面的 SQL:

```
1.  SELECT
2.      ROW_NUMBER() OVER(ORDER BY SALARY) AS 序号,
3.      NAME AS 姓名,
4.      DEPT AS 部门,
5.      SALARY AS 工资
6.  FROM
7.  (
8.      --姓名     部门   工资
9.      VALUES
10.     ('张三','市场部',4000),
11.     ('赵红','技术部',2000),
12.     ('李四','市场部',5000),
13.     ('李白','技术部',5000),
14.     ('王五','市场部',NULL),
15.     ('王蓝','技术部',4000)
16.) AS EMPLOY(NAME,DEPT,SALARY);
17.
18.查询结果如下：
19.
20.序号         姓名         部门         工资
21.1       赵红         技术部      2000
22.2       张三         市场部      4000
23.3       王蓝         技术部      4000
24.4       李四         市场部      5000
25.5       李白         技术部      5000
26.6       王五         市场部      (null)
```

看到上面的 ROW_NUMBER() OVER()了吗？很多人非常不理解，怎么两个函数能这么写呢？甚至有人怀疑上面的 SQL 语句是不是真的能执行。其实，ROW_NUMBER 是个函数没错，它的作用从它的名字也可以看出来，就是给查询结果集编号。但是，OVER 并不是一个函数，而是一个表达式，它的作用是定义一个作用域（或者可以说是结果集），OVER 前面的函数只对 OVER 定义的结果集起作用。怎么样，不明白？没关系，我们后面还会详细介绍。

从上面的 SQL 我们可以看出，典型的 DB2 在线分析处理的格式包括两部分：**函数部分**和 **OVER 表达式部分**。那么，函数部分可以有哪些函数呢？如下：

```
1. ROW_NUMBER
2. RANK
3. DENSE_RANK
4. FIRST_VALUE
5. LAST_VALUE
6. LAG
7. LEAD
8. COUNT
9. MIN
10. MAX
11. AVG
12. SUM
```

上面这些函数的作用，我会在后面逐步给大家介绍，大家可以根据函数名猜测一下函数的作用。

假设我想在不改变上面语句的查询结果的情况下，追加对部门员工的平均工资和全体员工的平均工资的查询，怎么办呢？用通常的 SQL 很难查询，但是用 OLAP 函数则非常简单，如下 SQL 所示：

```
1. SELECT
2.     ROW_NUMBER() OVER() AS 序号,
3.     ROW_NUMBER() OVER(PARTITION BY DEPT ORDER BY SALARY) AS 部门序号,

4.     NAME AS 姓名,
5.     DEPT AS 部门,
6.     SALARY AS 工资,
7.     AVG(SALARY) OVER(PARTITION BY DEPT) AS 部门平均工资,
8.     AVG(SALARY) OVER() AS 全员平均工资
9. FROM
10. (
11.     --姓名     部门   工资
12.     VALUES
13.     ('张三','市场部',4000),
14.     ('赵红','技术部',2000),
15.     ('李四','市场部',5000),
16.     ('李白','技术部',5000),
17.     ('王五','市场部',NULL),
18.     ('王蓝','技术部',4000)
19. ) AS EMPLOY(NAME,DEPT,SALARY);
20.
```

```
21.
22. 查询结果如下：
23.
24. 序号        部门序号        姓名        部门        工资        部门平均工资
        全员平均工资
25. 1             1             张三        市场部      4000        4500
                4000
26. 2             2             李四        市场部      5000        4500
                4000
27. 3             3             王五        市场部      (null)      4500
                4000
28. 4             1             赵红        技术部      2000        3666
                4000
29. 5             2             王蓝        技术部      4000        3666
                4000
30. 6             3             李白        技术部      5000        3666
                4000
```

请注意序号和部门序号之间的区别，我们在查询部门序号的时候，在 OVER 表达式中多了两个子句，分别是 PARTITION BY 和 ORDER BY。它们有什么作用呢？在介绍它们的作用之前，我们先来回顾一下 OVER 的作用，还记得吗？

OVER 是一个表达式，它的作用是定义一个作用域（或者可以说是结果集），OVER 前面的函数只对 OVER 定义的结果集起作用。

ORDER BY 的作用大家应该非常熟悉，用来对结果集排序。PARTITION BY 的作用其实也很简单，和 GROUP BY 的作用相同，用来对结果集分组。

到此为止，大家应该对 OLAP 函数的套路有一定的了解和体会了吧。大家看一下上面 SQL 的结果集，发现王五的工资是 null，当我们按工资排序时，null 被放到最后，我们想把 null 放在前边该怎么办呢？使用 NULLS FIRST 关键字即可，默认是 NULLS LAST，请看下面的 SQL：

```
1.  SELECT
2.      ROW_NUMBER() OVER(ORDER BY SALARY desc NULLS FIRST) AS RN,
3.      RANK() OVER(ORDER BY SALARY desc NULLS FIRST) AS RK,
4.      DENSE_RANK() OVER(ORDER BY SALARY desc NULLS FIRST) AS D_RK,
5.      NAME AS 姓名,
6.      DEPT AS 部门,
7.      SALARY AS 工资
8.  FROM
9.  (
10.     --姓名     部门   工资
```

```
    VALUES
    ('张三','市场部',4000),
    ('赵红','技术部',2000),
    ('李四','市场部',5000),
    ('李白','技术部',5000),
    ('王五','市场部',NULL),
    ('王蓝','技术部',4000)
) AS EMPLOY(NAME,DEPT,SALARY);

查询结果如下：

RN   RK    D_RK      姓名        部门        工资
1      1      1       王五        市场部      (null)
2      2      2       李四        市场部      5000
3      2      2       李白        技术部      5000
4      4      3       张三        市场部      4000
5      4      3       王蓝        技术部      4000
6      6      4       赵红        技术部      2000
```

请注意 ROW_NUMBER 和 RANK 之间的区别，RANK 是等级，排名的意思，李四和李白的工资都是 5000，他们并列排名第二。张三和王蓝的工资都是 4000，怎么 RANK 函数的排名是第四，而 DENSE_RANK 的排名是第三呢？这正是这两个函数之间的区别。由于有两个第二名，所以 RANK 函数默认没有第三名。

现在又有个新问题，假设让你查询一下每个员工的工资以及工资小于他的所有员工的平均工资，该怎么办呢？怎么？没听明白问题？不要紧，请看下面的 SQL：

```
SELECT
    NAME AS 姓名,
    SALARY AS 工资,
    SUM(SALARY) OVER(ORDER BY SALARY NULLS FIRST ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS 小于本人工资的总额,
    SUM(SALARY) OVER(ORDER BY SALARY NULLS FIRST ROWS BETWEEN  CURRENT ROW AND UNBOUNDED FOLLOWING) AS 大于本人工资的总额,
    SUM(SALARY) OVER(ORDER BY SALARY NULLS FIRST ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS 工资总额1,
    SUM(SALARY) OVER() AS 工资总额2
FROM
(
    --姓名     部门   工资
    VALUES
    ('张三','市场部',4000),
    ('赵红','技术部',2000),
    ('李四','市场部',5000),
```

```
    ('李白','技术部',5000),
    ('王五','市场部',NULL),
    ('王蓝','技术部',4000)
) AS EMPLOY(NAME,DEPT,SALARY);
```

查询结果如下：

姓名	工资	小于本人工资的总额	大于本人工资的总额	工资总额 1	工资总额 2
王五	(null)	(null)	20000	20000	20000
赵红	2000	2000	20000	20000	20000
张三	4000	6000	18000	20000	20000
王蓝	4000	10000	14000	20000	20000
李四	5000	15000	10000	20000	20000
李白	5000	20000	5000	20000	20000

上面 SQL 中的 OVER 部分出现了一个 ROWS 子句，我们先来看一下 ROWS 子句的结构：

```
ROWS BETWEEN <上限条件> AND <下限条件>

其中"上限条件"可以是如下关键字：
UNBOUNDED PRECEDING
<number>  PRECEDING
CURRENT ROW

"下线条件"可以是如下关键字：
CURRENT ROW
<number> FOLLOWING
UNBOUNDED FOLLOWING
```

注意，以上关键字都是相对当前行的，UNBOUNDED PRECEDING 表示当前行前面的所有行，也就是说没有上限；*<number>* PRECEDING 表示从当前行开始到它前面的*<number>*行为止，例如，number=2，表示的是当前行前面的 2 行；CURRENT ROW 表示当前行。至于其它两个关键字，我想，不用我说，你也应该知道了吧。如果你还不明白，请仔细分析上面 SQL 的查询结果。

OVER 表达式还可以有个子句，那就是 RANGE，它的使用方式和 ROWS 十分相似，或者说一模一样，作用也差多不，不过有点区别，如下所示：

RANGE BETWEEN *<上限条件>* AND *<下限条件>*

其中的*<上限条件>*、*<下限条件>*和 ROWS 一模一样，如下的 SQL 演示它们之间的区别：

```
SELECT
    NAME AS 姓名,
    DEPT AS 部门,
    SALARY AS 工资,
    FIRST_VALUE(SALARY, 'IGNORE NULLS') OVER(PARTITION BY DEPT) AS 部门最低工资,
    LAST_VALUE(SALARY, 'RESPECT NULLS') OVER(PARTITION BY DEPT) AS 部门最高工资,
    SUM(SALARY) OVER(ORDER BY SALARY ROWS BETWEEN 1 PRECEDING  AND 1 FOLLOWING) AS ROWS,
    SUM(SALARY) OVER(ORDER BY SALARY RANGE BETWEEN 500 PRECEDING AND 500 FOLLOWING) AS RANGE
FROM
(
    --姓名    部门  工资
    VALUES
    ('张三','市场部',2000),
    ('赵红','技术部',2400),
    ('李四','市场部',3000),
    ('李白','技术部',3200),
    ('王五','市场部',4000),
    ('王蓝','技术部',5000)
) AS EMPLOY(NAME,DEPT,SALARY);

查询结果如下：

姓名        部门        工资        部门最低工资        部门最高工资        ROWS        RANGE
张三        市场部      2000        2000                4000                4400        4400
赵红        技术部      2400        2400                5000                7400        4400
李四        市场部      3000        2000                4000                8600        6200
```

```
27.李白        技术部     3200        2400              5000              1020
   0      6200
28.王五        市场部     4000        2000              4000              1220
   0      4000
29.王蓝        技术部     5000        2400              5000              9000
           5000
```

上面 SQL 的 **RANGE** 子句的作用是定义一个工资范围，这个范围的上限是当前行的工资-500，下限是当前行工资+500。例如：李四的工资是 3000，所以上限是 3000-500=2500，下限是 3000+500=3500，那么有谁的工资在 2500-3500 这个范围呢？只有李四和李白，所以 RANGE 列的值就是 3000(李四)+3200(李白)=6200。以上就是 ROWS 和 RANGE 得区别。

上面的 SQL 还用到了 **FIRST_VALUE 和 LAST_VALUE** 两个函数，它们的作用也非常简单，用来求 OVER 定义集合的最小值和最大值。值得注意的是这两个函数有个参数，**'IGNORE NULLS' 或 'RESPECT NULLS'**，它们的作用正如它们的名字一样，用来忽略 NULL 值和考虑 NULL 值。

还有两个函数我们没有介绍，**LAG 和 LEAD**，这两个函数的功能非常强大，请看下面 SQL：

```
1. SELECT
2.      NAME AS 姓名,
3.      SALARY AS 工资,
4.      LAG(SALARY,0) OVER(ORDER BY SALARY) AS LAG0,
5.      LAG(SALARY) OVER(ORDER BY SALARY) AS LAG1,
6.      LAG(SALARY,2) OVER(ORDER BY SALARY) AS LAG2,
7.      LAG(SALARY,3,0,'IGNORE NULLS') OVER(ORDER BY SALARY) AS LAG3,
8.      LAG(SALARY,4,-1,'RESPECT NULLS') OVER(ORDER BY SALARY) AS LAG4,
9.      LEAD(SALARY) OVER(ORDER BY SALARY) AS LEAD
10.FROM
11.(
12.     --姓名     部门   工资
13.     VALUES
14.     ('张三','市场部',2000),
15.     ('赵红','技术部',2400),
16.     ('李四','市场部',3000),
17.     ('李白','技术部',3200),
18.     ('王五','市场部',4000),
19.     ('王蓝','技术部',5000)
20.) AS EMPLOY(NAME,DEPT,SALARY);
21.
22.查询结果如下：
```

```
23.
24.姓名        工资        LAG0        LAG1        LAG2        LAG3        LAG4
      LEAD
25.张三        2000        2000        (null)      (null)      0           -1
       2400
26.赵红        2400        2400        2000        (null)      0           -1
       3000
27.李四        3000        3000        2400        2000        0           -1
       3200
28.李白        3200        3200        3000        2400        2000        -1
       4000
29.王五        4000        4000        3200        3000        2400        2000
       5000
30.王蓝        5000        5000        4000        3200        3000        2400
       (null)
```

我们先来看一下 LAG 和 LEAD 函数的声明，如下：

LAG(表达式或字段, 偏移量, 默认值, IGNORE NULLS 或 RESPECT NULLS)

LAG 是向下偏移，LEAD 是想上偏移，大家看一下上面 SQL 的查询结果就一目了然了。

到此为止，有关 DB2 OLAP 函数的所有知识都介绍给大家了，下面我们再次回顾一下 DB2 在线分析处理 的组成部分，如下：

函数 OVER(PARTITION BY 子句 ORDER BY 子句 ROWS 或 RANGE 子句)

要想熟练掌握这些知识还需要一定的时间和练习，一旦你掌握了，你将拥有一项绝世武学，可以纵横 DB2。

DB2 分页查询

提起分页查询，除了那些还不知道什么是分页的人，大多数人的都会想到一个词，那就是 LIMIT，不过很可惜，DB2 不支持这个关键字，那么 DB2 的分页查询到底该怎么写呢？只要你学会了 OLAP 函数，分页查询是非常简单的。即使你不会 OLAP 函数，按照下面的 SQL 照猫画虎也可以，如下：

```
SELECT * FROM
(
SELECT B.*, ROWNUMBER() OVER() AS RN FROM
(
SELECT * FROM <TABLE_NAME>
) AS B
)AS A WHERE A.RN BETWEEN <START_NUMBER> AND <END_NUMBER>;
```

其中，尖括号中的内容是需要你根据实际情况替换的。至于其中的 ROWNUMBER() OVER() 是什么意思，我想，如果你会使用 OLAP 函数，不用我解释你也知道是什么意思；相反，三言两语也解释不清楚。

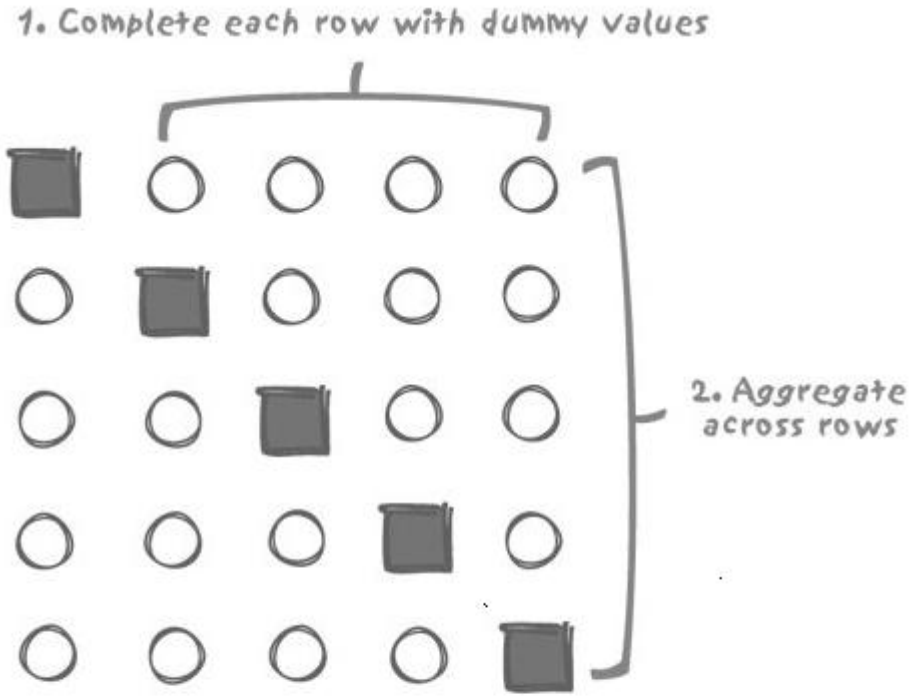
DB2 行转列

在网上看到这样一个问题：（问题地址：
http://www.mydb2.cn/bbs/read.php?tid=1297&page=e&#a）

```
1. 班级   科目    分数
2. 1      语文    8800
3. 1      数学    8420
4. 1      英语    7812
5. ……
6. 2      语文    8715
7. 2      数学    8511
8. 2      英语    8512
9. ……
10.
11.
12.要求转换成下面这样的结果
13.班级      语文      数学      英语
14.1         8800      8420      7812
15.2         8715      8511      8512
```

这是一个非常经典的 4 属性的表设计模式，顾名思义，这样的表一般有四列，分别是：entity_id, attribute_name, attribute_type, attribute_value ，这样的设计使我们添加字段非常容易，如：我们想添加一个物理成绩是非常简单的，我们只要向表中插入一条记录即可。但是，这样的设计有一个非常严重的问题，那就是：查询难度增加，查询效率非常差。

要想实现上面的查询有一个原则，那就是：通过 case 语句创造虚拟字段，使结果集成为二维数组，然后应用聚合函数返回单一记录。怎么样？不理解，仔细看看下面的图和分析下面的语句你就理解了。




```
1. create table score
2. (
3.     banji integer,
4.     kemu varchar(10),
5.     fengshu integer
6. )
7. go
8.
9. insert into score values
10.(1, '语文', 8800),
11.(1, '数学', 8420),
12.(1, '英语', 7812),
13.(2, '语文', 8715),
14.(2, '数学', 8511),
15.(2, '英语', 8512)
16.go
17.
18.select banji,
```

```
        max(yuwen)          语文,
        max(shuxue)         数学,
        max(yingyu)         英语
from
     (select  banji,
              case kemu
                when '语文' then fengshu
                else 0
              end                          yuwen,
              case kemu
                when '数学' then fengshu
                else 0
              end                          shuxue,
              case kemu
                when '英语' then fengshu
                else 0
              end                          yingyu
      from score
      ) as inner
group by inner.banji
order by 1
go
```

你可能正在感叹，这样的解决方案是多么的巧妙，可惜不是我想出来的，在这里，我也不敢把大师的思想据为己有，以上思想来自<SQL 语言艺术>的第 11 章，想了解更全面的信息，大家可以参考。

一个类似行转列的问题

在网上看到这样一个问题：（问题地址：http://bbs2.chinaunix.net/thread-1454869-1-7.html）

```
有一个表 T，内容如下：
ID        A          B
1         1          a
2         2          b
3         1          c
4         1          d
5         3          e
6         3          f

要求检索进行信息组合，查询结果如下：
RA            RB
1             a,c,d
2             b
3             e,f
```

这个问题类似 db 行转列问题，可以用同样的思想解决，这里我就不介绍了，大家可以参考：**DB2 行转列**

对于这个问题，下面有一个高手给出了用递归的解决方案，sql 如下：

```
WITH T1(A,B,NUM) AS
(
    SELECT A,B,ROW_NUMBER() OVER(PARTITION BY A ORDER BY ID) FROM T

),
T2(RA,RB,NUM) AS
(
SELECT A,CHAR(B,10),NUM FROM T1 WHERE NUM = 1
UNION ALL
SELECT T2.RA,RTRIM(T2.RB)||','||T1.B,T1.NUM FROM T1 , T2 WHERE T1.NUM
 = T2.NUM + 1 AND T1.A = T2.RA
)
SELECT
    RA,RB FROM T2
WHERE
    NUM = ( SELECT MAX(NUM) FROM T2 TEMP WHERE TEMP.RA = T2.RA)
ORDER BY RA
```

这样的解决方案同样很巧妙，值得学习。

更多简单而实用的 DB2 SQL 语句

查看当前时间

VALUES CURRENT TIME;

查看当前日期

VALUES CURRENT DATE;

查看当前时间戳

VALUES CURRENT TIMESTAMP;

查看当前时区

VALUES CURRENT TIMEZONE;

查看用户

VALUES USER;

查看系统用户

VALUES SYSTEM_USER;

查看连接用户

VALUES SESSION_USER;

查看当前用户

VALUES CURRENT USER;

查看当前客户端的账户名称

VALUES CURRENT CLIENT_ACCTNG

查看当前客户端的应用程序名称

VALUES CURRENT CLIENT_APPLNAME

查看当前客户端的用户名称

VALUES CURRENT CLIENT_USERID

查看当前客户端的工作站名称

VALUES CURRENT CLIENT_WRKSTNNAME

查看当前 SCHEMA

VALUES CURRENT SCHEMA

查看当前数据库名

VALUES CURRENT SERVER

查看当前路径

VALUES CURRENT PATH

查看当前包路径

VALUES CURRENT PACKAGE PATH

查看锁超时时间

VALUES CURRENT LOCK TIMEOUT

查看查询优先级(0-9)

VALUES CURRENT QUERY OPTIMIZATION

查看 DB2 版本

SELECT * FROM SYSIBM.SYSVERSIONS

查看系统中有哪些表以及这些表的概要信息

SELECT * FROM SYSCAT.TABLES

查看某个表有哪些列以及这些列的概要信息

SELECT * FROM SYSCAT.COLUMNS WHERE TABNAME=<YOUR_TABLE_NAME
>
查看某个表的权限
SELECT * FROM SYSCAT.TABAUTH WHERE TABNAME=''

如何写出高效的 SQL

要想写出高效的 SQL 语句需要掌握一些基本原则，如果你违反了这些原则，一般情况下 SQL 的性能将会很差。

一：减少数据库访问次数

连接数据库是非常耗时的，虽然应用程序会采用连接池技术，但与数据库交互依然很耗时，这就要求我们尽量用一条语句干完所有的事，尤其要避免把 SQL 语句写在循环中，如果你遇到这样的人，应该毫不犹豫给他两个耳光。

二：避免在有索引的字段上使用函数

在索引字段上使用函数会使索引失效，我们可以通过其他方式避免使用函数，如：尽量避免在 SQL 语句的 WHERE 子句中使用函数

三：避免在 SQL 语句中使用过程逻辑

通常开发人员思考问题喜欢采用过程逻辑，而 SQL 语句操作的对象是集合，所以写 SQL 语句时时刻提醒自己不要采用过程逻辑，否则会写出非常拙劣的 SQL。

四：采用乐观式 SQL

通常，开发人员写程序时会先判断参数的有效性，然后执行一定的操作，而在访问数据库时，可以先执行 SQL，然后，判断影响的行数，这样可以减少和数据库的交互。

五：将排序操作放到最后

排序操作非常耗时，通常，我们应该把所有不必要的记录都剔除后在进行排序操作，如果能不排序，尽量不要排序。

DB2 特殊寄存器(Special Registers)

所谓的特殊寄存器，其实就是一些变量，这些变量显示了 DB2 的一些状态信息，我们可以查看所有这些变量，也可以更新其中的一部分变量。

查看方法：

```
1. SELECT CURRENT TIME FROM sysibm.sysdummy1;
2. VALUES CURRENT TIME;
```

更新方法：

```
1. SET CURRENT SCHEMA = 'DB2ADMIN';
```

以下是所有的特殊寄存器列表，关于其中每个变量的具体含义可以参考 db2 信息中心。

```
1. SPECIAL                                  REGISTER UPDATE    DATA-TYPE
2. ==================================     ========    ================
3. CURRENT CLIENT_ACCTNG                       no           VARCHAR(255)
4. CURRENT CLIENT_APPLNAME                     no           VARCHAR(255)
5. CURRENT CLIENT_USERID                       no           VARCHAR(255)
6. CURRENT CLIENT_WRKSTNNAME                   no           VARCHAR(255)
7. CURRENT DATE                                no           DATE
8. CURRENT DBPARTITIONNUM                      no           INTEGER
9. CURRENT DECFLOAT ROUNDING MODE              no           VARCHAR(128)
10. CURRENT DEFAULT TRANSFORM GROUP            yes          VARCHAR(18)
11. CURRENT DEGREE                             yes          CHAR(5)
12. CURRENT EXPLAIN MODE                       yes          VARCHAR(254)
13. CURRENT EXPLAIN SNAPSHOT                   yes          CHAR(8)
14. CURRENT FEDERATED ASYNCHRONY               yes          INTEGER
15. CURRENT IMPLICIT XMLPARSE OPTION           yes          VARCHAR(19)
16. CURRENT ISOLATION                          yes          CHAR(2)
17. CURRENT LOCK TIMEOUT                       yes          INTEGER
18. CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION       yes          VAR
    CHAR(254)
```

19.	CURRENT MDC ROLLOUT MODE	yes	VARCHAR(9)
20.	CURRENT OPTIMIZATION PROFILE	yes	VARCHAR(261)
21.	CURRENT PACKAGE PATH	yes	VARCHAR(4096)
22.	CURRENT PATH	yes	VARCHAR(2048)
23.	CURRENT QUERY OPTIMIZATION	yes	INTEGER
24.	CURRENT REFRESH AGE	yes	DECIMAL(20,6)
25.	CURRENT SCHEMA	yes	VARCHAR(128)
26.	CURRENT SERVER	no	VARCHAR(128)
27.	CURRENT TIME	no	TIME
28.	CURRENT TIMESTAMP	no	TIMESTAMP
29.	CURRENT TIMEZONE	no	DECIMAL(6,0)
30.	CURRENT USER	no	VARCHAR(128)
31.	SESSION_USER	yes	VARCHAR(128)
32.	SYSTEM_USER	no	VARCHAR(128)
33.	USER	yes	VARCHAR(128)

DB2 物化查询表

DB2 物化查询表 MQT（MATERIALIZED QUERY TABLES）存储了一个查询的结果，当我们查询相关表时，DB2 会自动决定是使用原表还是使用物化查询表。当数据库中有海量数据时，使用物化查询表可以极大的提高查询速度。但是，有一利就有一弊，维护物化查询表也是相当耗时的。所以，物化查询表广泛应用在数据仓库和海量数量的报表查询中，这类查询的特点是：数据量大、经常需要分组统计、数据不会频繁变更。正因为这些特点，在这些场合中物化查询表可以充分发挥它的优势。

```
1. 语法：
2. CREATE TABLE <table-name> AS
3. <select stmtement>
4. DATA INITIALLY DEFERRED
5. REFRESH [DEFERRED | IMMEDIATE]
6. [ENABLE QUREY OPTIMIZATION | DISABLE QUREY OPTIMIZATION]
7. [MAINTAINED BY [SYSTEM | USER | FEDERATED_TOOOL]]
8.
9. 示例：
10.CREATE TABLE emp_summary AS
11.(
12.          SELECT
13.                    workdept
14.                    ,COUNT(*) AS crows
15.                    ,SUM(empno) AS sumno
16.          FROM
17.                    employee
18.          GROUP BY workdept
19.)
20.DATA INITIALLY DEFERRED
21.REFRESH IMMEDIATE;
```

定义了物化查询表后，如果我们执行以下 SQL，DB2 优化器将使用 MQT

```
1. select workdept,avg(empno) from employee group by workdept
```

DB2 优化器将上面的 SQL 转化成下面这样

```
1. select workdept,sumno/crows from emp_summary
```

在定义物化查询表时，我们可以指定在原始表数据改变时，是立即刷新物化查询表（REFRESH IMMEDIATE）呢，还是延迟刷新（REFRESH DEFERRED ）；我们还可以指定，在适当的时候，允许优化器使用物化查询表（ENABLE QUREY OPTIMIZATION）

呢，还是禁止使用（DISABLE QUREY OPTIMIZATION]）；我们还可以指定，物化查询表是由系统维护（MAINTAINED BY SYSTEM）呢，还是由用户维护（MAINTAINED BY USER）。

如果我们将物化查询表定义为延迟刷新（REFRESH DEFERRED ），那么在使用物化查询表之前，我们必须使用 REFRESH TABLE 语句刷新它。如果定义为由用户负责维护物化查询表时，用户可以对物化查询表进行 insert update delete 等操作，此时，物化查询表将不能 REFRESH 了。

维护物化查询表是相当耗时的，为了提高维护效率，我们可以给延迟刷新（REFRESH DEFERRED）的物化查询表定义一个 staging 表。staging 表用来对物化查询表执行增量刷新，当刷新完成时，staging 表就会被删除。对于上面定义的物化查询表，我们可以定义如下 staging 表

```
1. CREATE TABLE emp_summary_st
2. (
3.          workdept,
4.          crows,
5.          sumno,
6.          GLOBALTRANSID,
7.          GLOBALTRANSTIME
8. )FOR emp_summary PROPAGATE IMMEDIATE;
```

PROPAGATE IMMEDIATE 子句表示，原始表做出的任何更改，都将被累积在 staging 表中。GLOBALTRANSID 表示每个被传播的行对应的全局事务 ID）。GLOBALTRANSTIME 表示事务的时间戳。taging 表创建后，处于检查暂挂状态，我们可以使用 SET INTEGRITY 语句将表设置为正常状态，这时候，我们就可以使用 staging 表来刷新物化查询表了。

```
1. SET INTEGRITY FOR emp_summary_st STAGING IMMEDIATE UNCHECKED;
2. REFRESH TABLE emp_summary;
```

更多细节请参考 DB2 信息中心。

第二部分 SQL PL 简介

SQL(Structured Query Language)，也就是结构化查询语言，它被设计用来操作集合的，是非过程化的语言。随着应用程序的发展，业务逻辑越来越复杂，传统的 SQL 已经不能满足人们的要求，于是人们对 SQL 进行了扩展，使它具有了过程化的逻辑，即：SQL PL。SQL PL 的全称是 SQL Procedural Language， 它是 SQL Persistent Stored Module 语言标准的一个子集。该标准结合了 SQL 访问数据的方便性和编程语言的流控制。通过 SQL PL 当前的语句集合和语言特性，可以用 SQL 开发综合的、高级的程序，例如函数、存储过程和触发器。这样便可以将业务逻辑封装到易于维护的数据库对象中，从而提高 数据库应用程序的性能。本系列文章将会给大家介绍 SQL PL 的语言特性。

数据类型和变量

一：数据类型

```
Data Types
     |---Numeric
     |      |---Integer
     |      |      |---SMALLINT
     |      |      |---INTEGER
     |      |      |---BIGINT
     |      |---DECIMAL
     |      |---Floating Point
     |      |      |---REAL
     |      |      |---DOUBLE
     |
     |---String
     |      |---Character String
     |      |         |---Single Byte
     |      |         |        |---CHAR
     |      |         |        |---VARCHAR
     |      |         |        |---LONG VARCHAR
     |      |         |        |---CLOB
     |      |         |---Double Byte
     |      |         |        |---GRAPHIC
     |      |         |        |---VARGRAPHIC
     |      |         |        |---LONG VARGRAPHIC
     |      |         |        |---DBCLOB
     |      |---Binary String---BLOB
     |
     |---Datetime
     |      |---DATE
     |      |---TIME
     |      |---TIMESTAMP
     |
     |---XML
```

二：声明变量

```
语法：
DECLARE <variable-name> <data-type> <DEFAULT constant>
```

```
3.
4.
5. 示例：
6. DECLARE x, y INT DEFAULT 0;
7. DECLARE myname VARCHAR(10);
8. DECLARE z DECIMAL(9,2) DEFAULT 0.0;
```

三：赋值

```
1. 方法 1：使用 SET 语句
2. 方法 2：使用 VALUES INTO 语句
3. 方法 3：使用 SELECT INTO 语句
4.
5.
6. 示例：
7. SET x=10;
8. SET y=(SELECT SUM(c1) from T1);
9.
10. VALUES 10 INTO x;
11.
12. SELECT SUM(c1) INTO y from T1
```

四：会话全局变量

DB2 支持会话全局变量。它与一个特定的会话相关联，它对于这个会话中的每个存储过程都是全局的，会话全局变量是在存储过程之外声明的。

```
1. 语法：
2. CREATE VARIABLE var_name DATATYPE [DEAFULT value];
3.
4. 示例：
5. CREATE VARIABLE myvar INTEGER default 0;
```

数组

DB2 从 9.5 开始支持数组。可以在存储过程和应用程序中使用数组，但不能在定义表的时候使用数组。

一：定义数组

```
1. 语法：
2. CREATE TYPE <array-type-name> AS <data-type> ARRAY[integer-constant]
3.
4. 例子：
5. CREATE TYPE nar as INTEGER ARRAY[100];
6. CREATE TYPE mynames as VARCHAR(30) ARRAY[];
```

二：声明数组

```
1. --语法：
2. DECLARE <array-name> <array-type-name>
3.
4. --例子：
5. DECLARE mynumber nar;
6. DECLARE nameArr  mynames;
```

三：赋值

```
1. 方法 1：使用 SET 语句
2. 方法 2：使用 VALUES INTO 语句
3. 方法 3：使用 SELECT INTO 语句
4. 方法 4：使用 ARRAY 构造函数
```

四：操作数组的函数

```
1. ARRAY_DELETE：删除数组元素
2. TRIM_ARRAY：从右开始删除指定数目个元素
3. ARRAY_FIRST：返回数组中第一个元素
4. ARRAY_LAST：返回数组中最后一个元素
5. ARRAY_NEXT：返回数组下一个元素
6. ARRAY_PRIOR：返回数组前一个元素
7. ARRAY_VARIABLE：返回参数指定的元素
8. ARRAY_EXISTS：判断数组是否有元素
9. CARDINALITY：返回数组中元素的个数
10. MAX_CARDINALITY：返回数组中元素的个数
```

11. UNNEST：将数组转换为表

五：示例

```
--连接数据库
CONNECT TO SAMPLE!

--定义数组
CREATE TYPE INTARRAY AS INTEGER ARRAY[10]!

--创建存储过程
CREATE PROCEDURE TEST ()
BEGIN
    --声明数组
    DECLARE TESTARR INTARRAY;

    --赋值方式 1
    SET TESTARR=ARRAY[1,2,3,4,5,6,7,8,9,10];

    --赋值方式 2
    SET TESTARR=ARRAY[VALUES (1),(2)];

    --赋值方式 3
    VALUES 1 INTO TESTARR[1];
    VALUES 2 INTO TESTARR[2];

    --赋值方式 4
    SET TESTARR[1] =1;

    --赋值方式 5
    SELECT SUM(NUM) INTO TESTARR[1] FROM (VALUES (1),(2)) AS TEMP(NUM);
END!

--调用存储过程
CALL TEST()!

--删除存储过程
DROP PROCEDURE TEST!

--关闭连接
CONNECT RESET!
```

六：调用

```
1. 将上面的代码保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令
2. db2 -td! -vf c:\test.sql
```

游标(Cursor)

游标(Cursor)有点像 Java 中的 List 类，用来定义一个集合，并允许遍历这个集合，从而使我们能够处理集合中的单个记录。典型的使用游标的过程如下：

一：声明游标

```
1. 语法：
2. DECLARE <cursor-name> CURSOR [WITHOUT HOLD | WITH HOLD]
3. [WITHOUT RETURN | WITH RETURN TO CALLER | WITH RETURN TO CLIENT]
4. FOR <select-statement>
5.
6. 示例：
7. DECLARE mycr CURSOR FOR
8. WITH RETURN TO CALLER
9. SELECT * FROM tablex;
10.
11. 说明：
12. WITHOUT RETURN/WITH return 选项指定游标的结果表是否用于作为从一个过程中返回的结果集。
13. WITH RETURN TO CALLER 选项指定将来自游标的结果集返回给调用者，后者可以是另一个过程或一个客户机应用程序。这是默认选项。
14. WITH RETURN TO CLIENT 选项指定将来自游标的结果集返回给客户机应用程序，绕过任何中间的嵌套过程。
15. WITHOUT HOLD 和 WITH HOLD 选项定义 COMMIT 操作之后的游标状态（open/close）。默认情况下为 WITHOUT HOLD。如果使用了 WITH HOLD 选项定义一个游标，那么在 COMMIT 操作之后，该游标保持 OPEN 状态。在 ROLLBACK 操作之后，所有游标都将被关闭。
```

二：打开游标：OPEN <cursor-name>

三：从游标中获取数据

```
1. 语法：
2. FETCH <field> FROM <cursor-name> INTO <variables>
3.
4. 示例：
5. FETCH mycr INTO :myvar
```

四：关闭游标：CLOSE <cursor-name>

注释

SQL PL 支持 2 种注释。

一：单行注释：使用两个-

```
--这是一个单行注释
```

二：多行注释：使用/*...*/

```
/*
这是一个多行注释
*/
```

复合语句(compound statement)

大多数程序设计语言使用大括号来定义复合语句，将大括号中的语句看做一个整体，SQL PL 也可以定义复合语句，格式如下：

```
语法：
label: BEGIN [ATOMIC | NOT ATOMIC]
--变量声明、过程逻辑等
END label

示例 1：
P1:BEGIN
    DECLARE var1 INT;
    DECLARE var1 INT;
END P1

示例 2：
BEGIN ATOMIC
    DECLARE var1 INT;
    DECLARE var1 INT;
END
```

当 BEGIN 后紧随 ATOMIC 关键字时，其封装的复合语句就被当作一个处理单元，也就是说，复合语句中的所有程序指令和语句都必须成功运行，以保证整个复合语句的成功运行。如果其中的一个语句发生错误，那么这整个存储过程所执行的结果都将回滚。

IF 语句

学过任何一种程序语言的人对 IF 语句应该都非常熟悉，下面我们看看 SQL PL 中 IF 语句的格式：

一：语法

```
IF <condition> THEN
        <SQL procedure statement>;
ELSEIF <condition> THEN
        <SQL procedure statement>;
ELSE
        <SQL procedure statement>;
END IF;
```

二：示例

```
--连接数据库
CONNECT TO SAMPLE!

--创建存储过程
CREATE PROCEDURE TESTIF (IN FRIEND VARCHAR(10), OUT MSG VARCHAR(30))

LANGUAGE SQL
BEGIN
    IF FRIEND='张三' THEN
        SET MSG='你好，张三';
    ELSEIF FRIEND='李四' THEN
        SET MSG='你好，李四';
    ELSE
        SET MSG='对不起，我不认识你';
    END IF;
END!

--调用存储过程
CALL TESTIF('张三',?)!

CALL TESTIF('王五',?)!

--删除存储过程
DROP PROCEDURE TESTIF!
```

```
--关闭连接
CONNECT RESET!
```

三：运行示例

```
1. 将上面的代码保存为 c:\test_if.sql,然后在 DB2 命令窗口中执行如下命令
2. db2 -td! -vf c:\test_if.sql
```

除了 IF 语句外，CASE 语句也可以实现分支判断的功能，详见：**SQL 中的分支判断（CASE 语句的使用）**

循环语句

SQL PL 支持的循环语句有 LOOP、WHILE、REPEAT 和 FOR。

一：WHILE 循环

1、语法

```
WHILE <condition>
DO
  <sql statements>;
END WHILE;
```

2、示例

```
--连接数据库
CONNECT TO SAMPLE!

--创建存储过程
CREATE PROCEDURE TESTWHILE (OUT NUM INT)
BEGIN
        DECLARE I INT DEFAULT 1;
        SET NUM=0;

        WHILE I<=10
        DO
                SET NUM=NUM+I;
                SET I=I+1;
        END WHILE;
END!

--调用存储过程
CALL TESTWHILE(?)!

--关闭连接
CONNECT RESET!
```

3、调用

```
将示例的内容保存为 c:\test_while.sql,然后在 DB2 命令窗口中执行如下命令
db2 -td! -vf c:\test_while.sql
```

二：FOR 循环

1、语法

```
FOR <loop_name> AS
    <sql statements>
DO
    <sql statements>;
END FOR;
```

2、示例

```
--连接数据库
CONNECT TO SAMPLE!

--创建存储过程
CREATE PROCEDURE TESTFOR (OUT NUM INT)
BEGIN
        SET NUM=0;

        FOR TEST AS
             SELECT I FROM
             (
                 VALUES
                 (1),
                 (2),
                 (3)
             ) AS TEMP(I)
        DO
             SET NUM=NUM+I;
        END FOR;
END!

--调用存储过程
CALL TESTFOR(?)!

--关闭连接
CONNECT RESET!
```

3、调用

```
将示例的内容保存为 c:\test_for.sql,然后在 DB2 命令窗口中执行如下命令
db2 -td! -vf c:\test_for.sql
```

三：LOOP 循环

1、语法

```
LABEL:LOOP
    <sql statements>;
    LEAVE LABEL;
END LOOP LABEL;
```

2、示例

```
--连接数据库
CONNECT TO SAMPLE!

--创建存储过程
CREATE PROCEDURE TESTLOOP (OUT NUM INT)
BEGIN
    DECLARE I INT DEFAULT 1;
    SET NUM=0;

    TEST_LOOP:LOOP
        SET NUM=NUM+I;
        SET I=I+1;
        IF I>10 THEN
            LEAVE TEST_LOOP;
        END IF;
    END LOOP TEST_LOOP;

END!

--调用存储过程
CALL TESTLOOP(?)!

--关闭连接
CONNECT RESET!
```

3、调用

```
将示例的内容保存为 c:\test_loop.sql,然后在 DB2 命令窗口中执行如下命令
db2 -td! -vf c:\test_loop.sql
```

四：REPEAT 循环

1、语法

```
REPEAT
    <sql statements>;
    UNTIL <condition>
END REPEAT;
```

2、示例

```
--连接数据库
CONNECT TO SAMPLE!

--创建存储过程
CREATE PROCEDURE TESTREPEAT (OUT NUM INT)
BEGIN
    DECLARE I INT DEFAULT 1;
    SET NUM=0;

    REPEAT
        SET NUM=NUM+I;
        SET I=I+1;
        UNTIL I>10
    END REPEAT;

END!

--调用存储过程
CALL TESTREPEAT(?)!

--关闭连接
CONNECT RESET!
```

3、调用

```
将示例的内容保存为 c:\test_repeat.sql,然后在 DB2 命令窗口中执行如下命令
db2 -td! -vf c:\test_repeat.sql
```

ITERATE、LEAVE、GOTO 和 RETURN

学习过任何一门编程语言的人对 continue 和 break 都不陌生。在 SQL PL 中，ITERATE 和 LEAVE 实现相同的作用。RETURN 的作用和大多数程序设计语言一样，用来将结果返回给它的调用者。另外，SQL PL 中还支持 GOTO 语句，通常不建议使用 GOTO，因为它会使程序混乱。

一：语法

```
ITERATE label
LEAVE label
GOTO label
RETURN <result>
```

二：示例

```
--连接数据库
CONNECT TO SAMPLE!
DROP PROCEDURE TEST!

--创建存储过程
CREATE PROCEDURE TEST (OUT NUM INT)
BEGIN
        DECLARE I INT DEFAULT 1;
        SET NUM=0;

        L1:WHILE 1=1
        DO
             SET NUM=NUM+I;
             SET I=I+1;

             IF I<10 THEN
                 ITERATE L1;
             END IF;

             IF I=10 THEN
                 GOTO L2;
             END IF;

             IF I>10 THEN
                 LEAVE L1;
             END IF;
```

```
        END WHILE;

        L2:RETURN 0;
END!

--调用存储过程
CALL TEST(?)!

--关闭连接
CONNECT RESET!
```

三：调用

```
将示例的内容保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令
db2 -td! -vf c:\test.sql
```

异常处理

首先，我们了解一下什么是 SQLCODE 与 SQLSTATE：

SQLCODE 是在每一条 SQL 语句执行后收到的代码。这些值的意义如下：

```
1. SQLCODE=0     该 SQL 执行成功
2. SQLCODE>0     该 SQL 执行成功，但返回了一个警告
3. SQLCODE<0     该 SQL 没有执行成功，并且返回了一个错误
4. SQLCODE=100   未找到指定值
```

SQLSTATE 是一个遵守 ISO/ANSI SQL92 标准的长为 5 个字符的字符串，这些值的意义如下：

```
1. SQLSTATE='00000'     成功
2. SQLSTATE '01XXX'     警告
3. SQLSTATE='02000'     未找到
4. 其他所有值为错误
5.
6. 我们也可以自定义 SQLSTATE，它必须是 5 个字符串且必须以数字 7、8 或 9 或者字母 I
   到 Z 开始。
```

SQLSTATE 是标准，它在各 RDBMS 间是相同的，一般定义的比较笼统；

SQLCODE 在各个 RDBMS 是不同的，它比 SQLSTATE 更具体。**通常，几个**

SQLCODE 可能对应一个 **SQLSTATE**。

值得注意的是，要在 SQL PL 中使用 SQLCODE 和 SQLSTATE，我们必要先声明它们，如下：

```
1. DECLARE SQLCODE INT DEFAULT 0;
2. DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

下面我们将比较 JAVA 的异常处理机制和 SQL PL 的异常处理机制。

在 JAVA 中，我们可以通过继承 Exception 类来定义自己的异常类，在 SQL PL 中，我们可以给特定的 SQLSTATE 声明一个自定义的名称，可以在后面的 SQL 中使用它。

例如，SQLSTATE='01004'表示字符串数据被截断，如下的语句可以将该 SQLSTATE 命名为 trunc：

```
语法：
DECLARE <condition-name> CONDITION FOR SQLSTATE <string-constant>;

示例：
DECLARE truc CONDITION FOR SQLSTATE '01004';

以下是预定义的 SQLSTATE 名称，无需定义就可以在 SQL PL 中使用：
SQLSTATE='01XXX'    SQLWARNING
SQLSTATE='02000'    NOT FOUND
其他 SQLSTATE         SQLEXCEPTION
```

在 JAVA 中，我们可以通过 throw 语句显示抛出异常，在 SQL PL 中 SIGNAL 语句可以实现同样的功能。

```
一：语法
SIGNAL SQLSTATE <string-constant> SET MESSAGE_TEXT=<variable-name or string-constant>;


二：示例
--连接数据库
CONNECT TO SAMPLE!

--删除存储过程
DROP PROCEDURE TESTSIGNAL!

--创建存储过程
CREATE PROCEDURE TESTSIGNAL (IN FRIEND VARCHAR(10))
LANGUAGE SQL
BEGIN
    --声明异常
    DECLARE myexcept CONDITION FOR SQLSTATE '70000';

    --对输入参数进行验证
    IF FRIEND='李四' THEN
        SIGNAL myexcept SET MESSAGE_TEXT = '输入参数不能为李四';
    END IF;

    --对输入参数进行验证
    IF FRIEND='张三' THEN
        SIGNAL SQLSTATE '70000' SET MESSAGE_TEXT = '输入参数不能为张三';
```

```
    END IF;
END!

--调用存储过程
CALL TESTSIGNAL('张三')!

CALL TESTSIGNAL('李四')!

CALL TESTSIGNAL('王五')!

--关闭连接
CONNECT RESET!


三：调用
将上面的代码保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令
db2 -td! -vf c:\test.sql
```

在 JAVA 中，当程序出现运行时错误时，程序将跳转到异常处理模块。在 SQL PL 中也可以定义异常处理模块。

通常，当 SQL 在运行中出现错误时，SQL 就会终止并返回客户端一个错误消息。我们也可以给一个特定类型的错误定义一个处理程序，这样，当 SQL 在运行中出现这类错误时，程序就是跳转到该异常处理模块。

```
一：语法
DECLARE [CONTINUE | EXIT | UNDO] HANDLER FOR <condition>
<sql-procedure-statement>

说明：
CUNTINUE 表示，当抛出异常后，由对应的异常处理器解决异常，工作流会继续执行抛出异常语句的下一个语句。
EXIT 表示，当抛出异常后，相应的异常处理器解决该异常，工作流会直接到程序的末尾。
UNDO 表示,当抛出异常后，对应的异常处理器解决此异常情况，工作流直接到达程序的末尾并且撤销所有已实现的操作，或者回滚所有已执行的语句。


二：示例
--连接数据库
CONNECT TO SAMPLE!

--删除存储过程
DROP PROCEDURE TEST!

--创建存储过程
```

```
CREATE PROCEDURE TEST (IN FRIEND VARCHAR(10), OUT state_out CHAR(5),OUT code_out INT)
LANGUAGE SQL
BEGIN
    --声明 SQLCODE 和 SQLSTATE
    DECLARE SQLCODE INT DEFAULT 0;
    DECLARE SQLSTATE CHAR(5) DEFAULT '00000';

    --声明异常
    DECLARE myexcept CONDITION FOR SQLSTATE '70000';

    --定义异常处理程序
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        SELECT SQLSTATE, SQLCODE INTO state_out, code_out FROM SYSIBM.SYSDUMMY1;

    --对输入参数进行验证
    IF FRIEND='张三' THEN
        SIGNAL myexcept SET MESSAGE_TEXT = '输入参数不能为张三';
    END IF;
END!

--调用存储过程
CALL TEST('张三', ?, ?)!

--关闭连接
CONNECT RESET!
```

三：调用

将上面的代码保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令

```
db2 -td! -vf c:\test.sql
```

GET DIAGNOSTIC 语句

GET DIAGNOSTIC 语句用于获取前面执行的 SQL 语句的相关信息，它可以获取以下信息：

```
1、前面执行的 SQL 语句处理的行数
2、前面执行的 SQL 语句返回的 DB2 错误或警告消息文本
3、与前一个 CALL 语句相关联的过程返回的状态值
```

一：语法：

```
GET DIAGNOSTICS <sql-variable-name>=[ROW_COUNT | DB2_RETURN_STATUS | <
condition-information>

condition-information:
EXCEPTION 1 <sql-variable-name>=[MESSAGE_TEXT | DB2_TOKEN_STRING]
```

二：示例：

```
--连接数据库
CONNECT TO SAMPLE!

--创建测试表
CREATE TABLE TEST (A CHAR(1))!

--创建存储过程 1
CREATE PROCEDURE TESTPROC1 (IN CH1 CHAR(2))
BEGIN
    DECLARE SQLCODE INTEGER DEFAULT 0;
    DECLARE SQLSTATE CHAR(5) DEFAULT '';
    DECLARE ERR_MSG VARCHAR(70);
    DECLARE R_STATE INTEGER DEFAULT 0;

    --声明异常处理模块
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
        --获取前面执行的 SQL 语句返回的错误或警告消息
        GET DIAGNOSTICS EXCEPTION 1 ERR_MSG=MESSAGE_TEXT;
        SET R_STATE=1;
    END;
```

```
22.
23.    INSERT INTO TEST VALUES (CH1);
24.    RETURN R_STATE;
25. END!
26.
27. --创建存储过程 2
28. CREATE PROCEDURE TESTPROC (OUT R_COUNT INTEGER, OUT R_STATE INTEGER)

29. P1:BEGIN
30.     SET R_COUNT=0;
31.     SET R_STATE=0;
32.
33.     --向 TEST 表插入数据
34.     INSERT INTO TEST (A) VALUES (1),(2);
35.
36.     --获取前面执行的 SQL 语句处理的行数
37.     GET DIAGNOSTICS R_COUNT=ROW_COUNT;
38.
39.     --调用 TESTPROC1
40.     CALL TESTPROC1('AB');
41.
42.     --获取前一个存储过程返回的状态值
43.     GET DIAGNOSTICS R_STATE = DB2_RETURN_STATUS;
44. END P1!
45.
46. --调用存储过程 2
47. CALL TESTPROC(?,?)!
48.
49. --删除存储过程
50. DROP PROCEDURE TESTPROC1!
51. DROP PROCEDURE TESTPROC!
52.
53. --删除测试表
54. DROP TABLE TEST!
55.
56. --关闭连接
57. CONNECT RESET!
```

三：运行示例：

```
1.  将上面的代码保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令
2.  db2 -td! -vf c:\test.sql
```

动态 SQL(Dynamic SQL)

动态 SQL 是在程序运行时构造的，要执行**单条 SQL**，使用 EXECUTE IMMEDATE 语句；当**批量执行 SQL** 时，先使用 PREPARE 语句构造 SQL，然后使用 EXECUTE 语句执行。

一：**Prepare 语句**：用来构造批量 SQL

```
1. 语法：
2. PREPARE <sql-statement> [OUTPUT] INTO <result> [INPUT INTO] <input> FR
   OM <variable>
```

二：**DESCRIBE 语句**：获取表、SQL 等数据库对象的描述信息

```
1. 语法：
2. >>-DESCRIBE-------------------------------------------------------->
3.       .-OUTPUT-.

4. >--+-+--------+--+-select-statement---------+-------------------------
   -------+->< 
5.     |             +-call-statement-----------+
            |
6.     |             '-XQUERY--XQuery-statement-'
            |
7.     '-+-TABLE-----------------------------------+--table-name--+--------
   -----+-'
8.       +-+------------------+--INDEXES FOR TABLE-+              '-SHOW DE
   TAIL-'
9.       | +-RELATIONAL DATA-+                    |

10.      | +-XML DATA--------+                    |

11.      | '-TEXT SEARCH-----'                    |

12.      '-DATA PARTITIONS FOR TABLE--------------'
```

三：**Execute 语句**：用来执行批量 SQL，不能执行 select 语句

```
1. 语法：
2. EXECUTE <statement-name> [INTO <result-variable>] [USING <input-
   variable> [,<input-variable>,...] ]
```

四：**Execute Immediate** 语句：用来执行单条 SQL 语句，不能执行 select 语句

```
1. 语法：
2. EXECUTE IMMEDIATE <sql-statement>
```

五：示例

```
1. --连接数据库
2. CONNECT TO SAMPLE!
3.
4. --创建表
5. CREATE TABLE TESTTB(A INT, B INT)!
6.
7. --删除存储过程
8. DROP PROCEDURE TEST!
9.
10. --创建存储过程
11. CREATE PROCEDURE TEST (IN V1 INT, IN V2 INT)
12. BEGIN
13.         DECLARE STMT VARCHAR(50);
14.         DECLARE ST STATEMENT;
15.
16.         --静态 SQL
17.         INSERT INTO TESTTB VALUES (1,1);
18.
19.         --动态 SQL(单条语句)
20.         SET STMT='INSERT INTO TESTTB VALUES (' || V1 || ',' || V2 || ')
    ';
21.         EXECUTE IMMEDIATE STMT;
22.
23.         --动态 SQL(批量执行)
24.         SET STMT = 'INSERT INTO TESTTB VALUES (?, ?)';
25.         PREPARE ST FROM STMT;--编译 SQL
26.         EXECUTE ST USING V1, V1;--绑定参数并执行 SQL
27.         EXECUTE ST USING V2, V2;
28. END!
29.
30. --调用存储过程
31. CALL TEST(2,3)!
32.
33. --查询 TESTTB 表
34. SELECT * FROM TESTTB!
35.
36. --删除表
```

```
37. DROP TABLE TESTTB!
38.
39. --关闭连接
40. CONNECT RESET!
```

内联 SQL PL(Inline SQL PL)

通常 SQL PL 只能使用在存储过程、触发器、用户自定义函数中，但是有一部分 SQL PL 也可以直接在命令行编辑器或脚本中使用，它们是：

```
1. DECLARE <variable>
2. SET
3. CASE
4. FOR
5. GET DIAGNOSTICS
6. GOTO
7. IF
8. RETURN
9. SIGNAL
10. WHILE
11. ITERATE
12. LEAVE
```

以下 SQL PL 不能直接在命令行编辑器或脚本中使用，它们只能在存储过程、触发器、用户自定义函数中使用：

```
1. ALLOCATE CURSOR
2. ASSOCIATE LOCATORS
3. DECLARE <cursor>
4. DECLARE ...HANDLER
5. PREPARE
6. EXECUTE
7. EXECUTE IMMEDIATE
8. LOOP
9. REPEAT
10. RESIGNAL
11. CALL
12. COMMIT/ROLLBACK
```

示例：

```
1. --连接数据库
2. CONNECT TO SAMPLE!
3.
4. --内联 SQL PL
5. BEGIN
6.         DECLARE I INT DEFAULT 1;
```

```
        DECLARE NUM INT DEFAULT 0;
        SET NUM=0;

        WHILE I<=10
        DO
                SET NUM=NUM+I;
                SET I=I+1;
        END WHILE;
END!

--关闭连接
CONNECT RESET!
```

调用示例：

```
1. 将示例的内容保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令
2. db2 -td! -vf c:\test.sql
```

存储过程

一：简介

存储过程是一个能够封装 SQL 语句和业务逻辑的数据库应用对象，DB2 存储过程可以用以下语言来表达：SQL PL, C/C++, Java, Cobol, CLR(Common LanguageRuntime)支持的语言, OLE。

存储过程可以被客户机应用程序、其他存储过程、用户定义函数或触发器调用，在 DB2 v9.5 中，一次最多可以嵌套 64 个存储过程。

存储过程对于其安全性也很有帮助。例如，您可以限制用户只能通过存储过程访问表和视图；这样可以锁定数据库而防止用户存取无权操作的那部分数据。用户通过存储过程存取数据表或者视图时不需要显式赋予权限，而只需要得到运行存储过程的权限。

二：简单的例子：将下面的代码保存为c:\test.sql,然后在DB2 命令窗口中执行命令：db2 -td! -vf c:\test.sql

```
--连接数据库
CONNECT TO SAMPLE!

--创建存储过程
CREATE PROCEDURE TEST_PROC3(IN PAR1 INT, INOUT PAR2 DECIMAL, OUT PAR3
 VARCHAR(20))
LANGUAGE SQL
DYNAMIC RESULT SETS 0
SPECIFIC TEST_PROC3_UNIQUE_NAME
P1: BEGIN ATOMIC
    SET PAR3='THIS IS TEST_PROC3 PROCEDURE!!!';
END P1!

--调用存储过程
CALL TEST_PROC3(2, 3.5, ?)!

--删除存储过程
DROP SPECIFIC PROCEDURE TEST_PROC3_UNIQUE_NAME!

--关闭连接
CONNECT RESET!
```

三：语法

```
语法：
CREATE PROCEDURE <schema-name>.<procedure-name> (参数) [属性] <语句>

--参数：SQL PL 存储过程中有三种类型的参数：
    IN：输入参数(默认值,也可以不指定)
```

```
6.      OUT：输出参数
7.      INOUT：输入和输出参数
8.
9.
10.--属性
11.1、LANGUAGE SQL
12.指定存储过程使用的语言。LANGUAGE SQL 是其默认值。还有其它的语言供选择，比如 Java 或者 C，可以将这一属性值分别设置为 LANGUAGE JAVA 或者 LANGUAGE C。
13.
14.2、DYNAMIC RESULT SETS <n>
15.如果您的存储过程将返回 n 个结果集，那么需要填写这一选项。
16.
17.3、SPECIFIC my_unique_name
18.赋给存储过程一个唯一名称，如果不指定，系统将生成一个惟一的名称。一个存储过程是可以被重载的，也就是说许多个不同的存储过程可以使用同一个名字，但这些存储过程所包含的参数数量不同。通过使用 SPECIFIC 关键字，您可以给每一个存储过程起一个唯一的名字，这可以使得我们对于存储过程的管理更加容易。例如，要使用 SPECIFIC 关键字来删除一个存储过程，您可以运行这样的命令：DROP SPECIFIC PROCEDURE。如果没有使用 SPECIFIC 这个关键字，您将不得不使用 DROP PROCEDURE 命令，并且指明存储过程的名字及其参数，这样 DB2 才能知道哪个被重载的存储过程是您想删除的。
19.
20.4、SQL 访问级别
21.NO SQL：存储过程中不能有 SQL 语句
22.CONTAINS SQL：存储过程中不能有可以修改或读数据的 SQL 语句
23.READS SQL：存储过程中不能有可以修改数据的 SQL 语句
24.MODIFIES SQL：存储过程中的 SQL 语句既可以修改数据，也可以读数据
25.    默认值是 MODIFIES SQL，一个存储过程不能调用具有更高 SQL 数据访问级别的其他存储过程。例如，被定义为 CONTAINS SQL 的存储过程可以调用被定义为 CONTAINS SQL 或 NO SQL 的存储过程。但是这个存储过程不能调用被定义为 READS SQL DATA 或 MODIFIES SQL 的其他存储过程。
26.
27.
28.--语句
29.可以是一条单独的语句或者是一组由 BEGIN [ATOMIC] ... END 复合语句
```

四：调用存储过程

使用 CALL 语句调用用存储过程，所有的参数都必须提供给 CALL 语句，输出参数用问号来设置

五：查询数据库中已经定义的存储过程

```
1. SELECT * FROM SYSCAT.PROCEDURES
```

在存储过程之间传递数据

存储过程之间可以相互调用，如何在它们之间传递数据呢？有以下 3 种方法。

一：通过输入输出参数传递数据

这是最简单的一种方式，不过值得注意的是，存储过程中的 RETURN 语句和其它程序设计语言中的 return 语句含义是不同的。存储过程中的 return 语句放回 sqlcode 值。

二：通过会话全局变量传递数据

```
--连接数据库
CONNECT TO SAMPLE!

--创建会话全局变量
CREATE VARIABLE MYVAR INTEGER DEFAULT 0!

--创建存储过程 1
CREATE PROCEDURE TESTPROC1 ()
BEGIN
    SET MYVAR=10;
END!

--创建存储过程 2
CREATE PROCEDURE TESTPROC2 (OUT MSG CHAR(20))
BEGIN
    CALL TESTPROC1();

    IF MYVAR=10 THEN
        SET MSG='测试成功';
    ELSE
        SET  MSG='测试失败';
    END IF;
END!

--调用存储过程 2
CALL TESTPROC2(?)!

--删除存储过程
DROP PROCEDURE TESTPROC1!
DROP PROCEDURE TESTPROC2!

--关闭连接
CONNECT RESET!
```

三：在存储过程之间传递游标

要想在一个存储过程中使用另一个存储过程的结果集，需要以下步骤：
1、声明一个结果集定位符：DECLARE <rs_locator_var> RESULT_SET_LOCATOR VARYING;
2、将这个结果集定位符与调用者过程相关联：ASSOCIATE RESULT SET LOCATOR(<rs_locator_var>) WITH PROCEDURE <proc_called>;
3、分配从调用过程指向结果集的游标：ALLOCATE <cursor> CURSOR FOR RESULT SET <rs_locator_var>;

```
--连接数据库
CONNECT TO SAMPLE!

--创建存储过程 1
CREATE PROCEDURE TESTPROC1 ()
DYNAMIC RESULT SETS 1
P1:BEGIN
    --声明游标
    DECLARE MYCUR CURSOR WITH RETURN FOR
    SELECT * FROM
    (
        VALUES
        (1),
        (2),
        (3)
    );
    --打开游标
    OPEN MYCUR;
END P1!


--创建存储过程 2
CREATE PROCEDURE TESTPROC2 (OUT RES INTEGER)
BEGIN
    DECLARE SQLCODE INT DEFAULT 0;

    --声明一个结果集定位符
    DECLARE LOC1 RESULT_SET_LOCATOR VARYING;
    DECLARE X INT DEFAULT 0;

    SET RES=0;
    CALL TESTPROC1();

    --将这个结果集定位符与调用者过程相关联
    ASSOCIATE RESULT SET LOCATOR(LOC1) WITH PROCEDURE TESTPROC1;

```

```
37.     --分配从调用过程指向结果集的游标
38.     ALLOCATE C1 CURSOR FOR RESULT SET LOC1;
39.
40.     FETCH FROM C1 INTO X;
41.     WHILE SQLCODE = 0 DO
42.         SET RES = RES + X;
43.         FETCH FROM C1 INTO X;
44.     END WHILE;
45. END!
46.
47. --调用存储过程 2
48. CALL TESTPROC2(?)!
49.
50. --删除存储过程
51. DROP PROCEDURE TESTPROC1!
52. DROP PROCEDURE TESTPROC2!
53.
54. --关闭连接
55. CONNECT RESET!
```

迁移存储过程

如果我们想将 SQL 过程从一个服务器转移到另一个服务器，或者分享给他人又不想让他知道内容，那么，我们可以将存储过程的内容加密，可以通过 PUT ROUTINE 和 GET ROUTINE 命令来实现。

一：GET ROUTINE 命令

GET ROUTINE 是一个 DB2 命令，它从数据库中提取一个 SQL 过程，并将它转换成一个 SAR（SQL Archive）文件。

```
1. 语法：
2. GET ROUTINE INTO <file_name> FROM SPECIFIC PROCEDURE <routine_name> HIDE BODY
3.
```

二：PUT ROUTINE 命令

PUT ROUTINE 是一个 DB2 命令，它根据通过 GET ROUTINE 提取的 SAR 文件，在数据库中创建 SQL 过程。

```
1. 语法：
2. PUT ROUTINE FROM <file-name> OWNER <new-owner> USE REGISTERS
```

用户自定义函数

我们可以创建五种类型的用户自定义函数：

```
1、有源（或模板）
2、SQL 标量、表或行
3、外部标量
4、外部表
5、OLE DB 外部表
```

一：有源函数：Sourced Function

顾名思义，有源函数是从一个已经存在的函数中构造出来的函数。它通常应用于操作用户自定义数据类型。因为现有的函数和操作符(+ - * /)不支持用户自定义的数据类型，所以当我们需要操作用户自定义数据类型时，就可以根据现有函数定义一个有源函数。

1、语法：

```
CREATE FUNCTION [FunctionName] ( <<[ParameterName]> [InputDataType]
 ,...> )
    RETURNS [OutputDataType]
    <SPECIFIC [SpecificName]>
    SOURCE [SourceFunction] <([DataType] ,...)>
    <AS TEMPLATE>

说明：
FunctionName 指定要创建的有源函数的名称。
ParameterName 指定一个或多个函数参数的名称。
InputDataType 指定 ParameterName 所识别的参数所需的数据类型。
OutputDataType 指定函数返回的数据的类型。
SpecificName 指定分配给这个 UDF 的特定名称。这个名称可以用来引用或删除函数；但是，不能用来调用函数。
SourceFunction 指定用来创建这个有源函数的现有函数。
DataType 指定现有函数的每个参数期望接收的数据类型。
```

2、示例：

```
--连接数据库
CONNECT TO SAMPLE;

--定义美元数据类型
CREATE DISTINCT TYPE US_DOLLARS AS DEC(7,2) WITH COMPARISONS;

--定义表
CREATE TABLE CUSTOMERS
```

```
(
ID SMALLINT NOT NULL,
BALANCE US_DOLLARS NOT NULL
);

--插入值
INSERT INTO CUSTOMERS VALUES (1 ,111.11),(2 ,222.22);

--查询 SQL
SELECT ID,BALANCE*10 FROM CUSTOMERS;

--上面的语句会发生异常，原因操作符*不支持 US_DOLLARS，为了上面的语句能执行，定义下面的函数：
CREATE FUNCTION "*" (US_DOLLARS,INT)
RETURNS US_DOLLARS
SOURCE SYSIBM."*"(DECIMAL,INT);

--再次执行查询 SQL
SELECT ID,BALANCE*10 FROM CUSTOMERS;

--删除用户定义函数
DROP FUNCTION "*" (US_DOLLARS,INT);

--删除表
DROP TABLE CUSTOMERS;

--删除美元数据类型
DROP DISTINCT TYPE US_DOLLARS;

--关闭连接
CONNECT RESET;
```

3、运行示例：

```
将上面的代码保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令
db2 -tvf c:\test.sql
```

二：SQL 标量、表或行函数

标量函数只返回一个值，标量函数中不允许 INSERT、UPDATE、DELETE 语句。

表函数返回一个表或者若干行，在 FROM 子句调用它们。表函数与标量函数不同，它能够使用 INSERT、UPDATE、DELETE 语句。

1、语法

```
CREATE FUNCTION [FunctionName] ( <<[ParameterName]> [InputDataType] ,...> )
    RETURNS [[OutputDataType] |
       TABLE ( [ColumnName] [ColumnDataType] ,... ) |
       ROW ( [ColumnName] [ColumnDataType] ,... )]
    <SPECIFIC [SpecificName]>
    <LANGUAGE SQL>
    <DETERMINISTIC | NOT DETERMINISTIC>
    <EXTERNAL ACTION | NO EXTERNAL ACTION>
    <CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA>
    <STATIC DISPATCH>
    <CALLED ON NULL INPUT>
    [SQLStatements] | RETURN [ReturnStatement]

```

说明：

FunctionName 指定要创建的 SQL 函数的名称。

ParameterName 指定一个或多个函数参数的名称。

InputDataType 指定 ParameterName 所识别的参数所需的数据类型。

OutputDataType 指定函数返回的数据的类型。

ColumnName 指定函数返回的一列或多列的名称（如果此函数返回表或行的话）。

ColumnDataType 指定 ColumnName 所识别的列返回的数据类型。

SpecificName 指定分配给这个 UDF 的特定名称。这个名称可以用来引用或删除函数；但是，不能用来调用函数。

SQLStatements 指定在调用函数时执行的一个或多个 SQL 语句。这些语句组合成一个动态复合 SQL 语句。

ReturnStatement 指定用于返回调用函数的应用程序的 RETURN SQL 语句。（如果 SQL 函数体由动态复合语句组成，那么它必须包含至少一个 RETURN 语句；在调用函数时，必须执行一个 RETURN 语句。如果函数是表函数或行函数，那么只能包含一个 RETURN 语句，而且此语句必须是使用的最后一个语句）。

<LANGUAGE SQL> 指定函数是用 SQL 编写的。

<DETERMINISTIC | NOT DETERMINISTIC> 表示在用相同的参数值调用函数时，函数是否总是返回相同的标量值、表或行（DETERMINISTIC 代表确定性函数，NOT DETERMINISTIC 表示非确定性函数）。如果没有指定这两个子句，那么在用相同的参数值调用函数时函数可能返回不同的结果。

<EXTERNAL ACTION | NO EXTERNAL ACTION> 表示函数执行的操作是否会改变不由 DB2 管理的对象的状态（EXTERNAL ACTION 表示会改变，NO EXTERNAL ACTION 表示不改变）。外部操作包括发送电子邮件或在外部文件中写记录等。如果没有指定这两个子句，那么意味着函数可能执行某种外部操作。

<CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA> 表示在 UDF 体中编写的 SQL 语句的类型。有三个值可用：

CONTAINS SQL：UDF 体包含的可执行 SQL 语句既不读数据，也不修改数据。

READS SQL DATA：UDF 体包含的可执行 SQL 语句读数据，但是不修改数据。

MODIFIES SQL DATA：UDF 体包含的可执行 SQL 语句既可以读数据，也可以修改数据。

31. <STATIC DISPATCH> 表示在函数解析时 DB2 根据函数参数的静态类型（声明的类型）选择函数。
32. <CALLED ON NULL INPUT> 表示无论任何参数是否包含 **null** 值，都调用此函数。

2、示例

```
1.  --连接数据库
2.  CONNECT TO SAMPLE!
3.
4.  --创建标量函数
5.  CREATE FUNCTION CHECK_LEN(INSTR VARCHAR(50))
6.  RETURNS SMALLINT
7.  BEGIN ATOMIC
8.      IF INSTR IS NULL THEN
9.          RETURN NULL;
10.     END IF;
11.
12.     IF LENGTH(INSTR) < 6 THEN
13.         SIGNAL SQLSTATE '75001' SET MESSAGE_TEXT = 'INPUT STRING IS < 6';
14.     ELSEIF LENGTH(INSTR) < 7 THEN
15.         RETURN -1;
16.     END IF;
17.
18.     RETURN LENGTH(INSTR);
19. END!
20.
21. --测试标量函数
22. VALUES CHECK_LEN('WAVE')!
23. VALUES CHECK_LEN('12345678')!
24.
25. --删除标量函数
26. DROP FUNCTION CHECK_LEN!
27.
28.
29.
30. --创建表函数
31. CREATE FUNCTION TESTTF()
32. RETURNS TABLE (ID SMALLINT,NAME VARCHAR(9))
33. RETURN SELECT * FROM (VALUES (1,'张三'),(2,'李四'))!
34.
35. --测试表函数:需要应用 TABLE()函数和别名
36. SELECT * FROM TABLE(TESTTF()) AS TEMP!
37.
```

```
--删除表函数
DROP FUNCTION TESTTF!

--关闭连接
CONNECT RESET;
```

3、运行示例：

```
将上面的代码保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令
db2 -td! -vf c:\test.sql
```

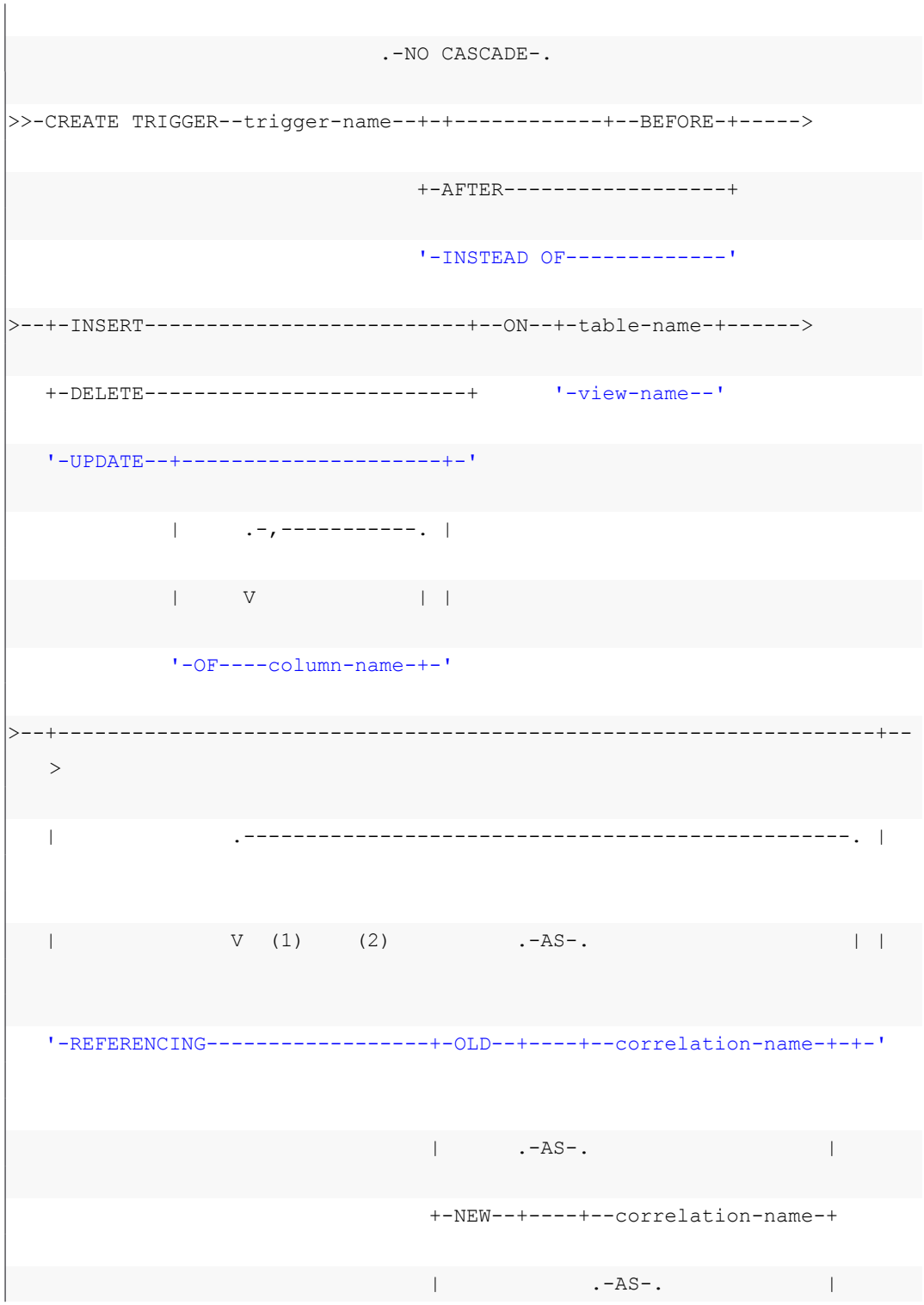
三：其他函数

外部标量函数、外部表函数和 Microsoft OLE DB 外部表函数，它们是用 C、C++ 或 Java™ 等外部高级编程语言编写的，这里就不做介绍了，更多细节请参考 DB2 信息中心 。

触发器(Trigger)

触发器和一个特定的表相关联，当向表中 INSERTT、UPDATE、DELETE 记录时，触发器会自动激活执行一些预定义的行为。

一：语法：



```
                                  +-OLD TABLE--+----+--identifier-+
                                  |            .-AS-.             |
                                  '-NEW TABLE--+----+--identifier-'
>--+-FOR EACH ROW--------------+--| triggered-action |--------->
   |  (3)                      |
   '--------FOR EACH STATEMENT-'
```

triggered-action

```
|--+-------------------------------------+---------------------->
   |  (4)                                |
   '--------WHEN--(--search-condition--)-'
```

说明：

一：触发器类型

1、前触发器(BEFORE):在 insert、update、delete 之前激活，主要用来验证、改变插入或更新的值。

2、后触发器(AFTER):在 insert、update、delete 之后激活，主要用来实现一些业务逻辑。

3、替代触发器(INSTEAD OF):在视图中定义，该触发器中定义的逻辑会替代视图中的触发 SQL 语句。

二：事件类型：DELETE、DELETE、UPDATE

三：对象类型

1、表：可以定义 BEFORE、AFTER 触发器

2、视图：可以定义 BNSTEAD OF 触发器

四：引用

1、OLD：引用改变之前的单行

2、NEW：引用改变之后的单行

3、OLD TABLE：引用改变之前的多行

4、NEW TABLE：引用改变之后的多行

五：应用范围

1、FOR EACH ROW：对改变的每行调用一次触发器

2、FOR EACH STATEMENT：每条 SQL 调用一次触发器

六：激活条件(WHEN)：只有符合条件时，触发器才会执行。

二：示例：

```
--连接数据库
CONNECT TO SAMPLE;

--创建表 USER
CREATE TABLE USER
(
    NAME VARCHAR(20) NOT NULL,--姓名
    SALARY FLOAT,--工资
    LASTUPDATEDDT TIMESTAMP--最后修改时间
);
```

```
--创建 USER 的历史表

CREATE TABLE USER_HIS

(

    NAME VARCHAR(20) NOT NULL,--姓名

    SALARY FLOAT,--工资

    LASTUPDATEDDT TIMESTAMP,--最后修改时间

    OPERATETIME TIMESTAMP,--操作时间

    OPERATETYPE CHAR(1)--操作类型(INSERT、UPDATE)

);



--USER 表的 LASTUPDATEDDT 总是由数据库自动赋值

CREATE TRIGGER USER_INS1

NO CASCADE BEFORE INSERT ON USER

REFERENCING NEW AS NNN

FOR EACH ROW

MODE DB2SQL

SET NNN.LASTUPDATEDDT = CURRENT TIMESTAMP;

2.
3.  CREATE TRIGGER USER_UPD1
4.  NO CASCADE BEFORE UPDATE ON USER
5.  REFERENCING NEW AS NNN
6.  FOR EACH ROW
7.  MODE DB2SQL
8.  SET NNN.LASTUPDATEDDT = CURRENT TIMESTAMP;
9.
10. --每当更改 USER 表时，自动记录到历史表中
11. CREATE TRIGGER USER_INS2
12. AFTER INSERT ON USER
13. REFERENCING NEW AS NNN
```

```
14. FOR EACH ROW
15. MODE DB2SQL
16. INSERT INTO USER_HIS VALUES (NNN.NAME,NNN.SALARY,NNN.LASTUPDATEDDT,CUR
    RENT TIMESTAMP,'I');
17.
18. CREATE TRIGGER USER_UPD2
19. AFTER UPDATE ON USER
20. REFERENCING NEW AS NNN
21. FOR EACH ROW
22. MODE DB2SQL
23. INSERT INTO USER_HIS VALUES (NNN.NAME,NNN.SALARY,NNN.LASTUPDATEDDT,CUR
    RENT TIMESTAMP,'U');
24.
25.
26. --USER_HIS 表不允许任何修改
27. CREATE TRIGGER USER_HIS_UPD1
28. NO CASCADE BEFORE UPDATE ON USER_HIS
29. FOR EACH ROW
30. WHEN (1=1) SIGNAL SQLSTATE VALUE '71001' SET MESSAGE_TEXT = 'USER_HIS
     表不允许更新';
31.
32. CREATE TRIGGER USER_HIS_DEL1
33. NO CASCADE BEFORE DELETE ON USER_HIS
34. FOR EACH ROW
35. WHEN (1=1) SIGNAL SQLSTATE VALUE '71001' SET MESSAGE_TEXT = 'USER_HIS
     表不允许删除';
36.
37.
38. --测试
39. INSERT INTO USER (NAME,SALARY,LASTUPDATEDDT) VALUES
40. ('张三',2000.0,'2009-8-18 12:25:00'),
41. ('李四',3500.0,'2009-12-18 12:25:00');
42. SELECT * FROM USER;
43. SELECT * FROM USER_HIS;
44. DELETE FROM USER_HIS;
45.
46. --删除
47. DROP TRIGGER USER_INS1;
48. DROP TRIGGER USER_UPD1;
49. DROP TRIGGER USER_INS2;
50. DROP TRIGGER USER_UPD2;
51. DROP TRIGGER USER_HIS_UPD1;
52. DROP TRIGGER USER_HIS_DEL1;
53. DROP TABLE USER;
```

```
54. DROP TABLE USER_HIS;
55.
56. --关闭连接
57. CONNECT RESET;
```

三：调用示例：

```
4.  将示例的内容保存为 c:\test.sql,然后在 DB2 命令窗口中执行如下命令
5.  db2 -tvf c:\test.sql
```

后记

恭喜你，你已经阅读完了本书。如果本书对你有所帮助，那么对我来说是件非常开心的事情。如果你觉得本书还不错，请将它分享给别人。

也许你已经注意到了，本书的名字《DB2 SQL 精萃》的“萃”字是个错别字，这是因为我女朋友叫萃萃，以此来感谢她对我的支持。

最后和大家分享一句话：**执着和放弃总在一念之间**。

尚波

2010-5-30 于大连