

SQL语句



单元目标

学完本单元，应该了解：

- DDL
- DML
- Select语句
- 连接查询
- 子查询
- 函数
- With表达式
- 分析函数

SQL发展历史

- 1970: E.J. Codd 发表了关系数据库理论;
 - 1974-79: IBM 以Codd的理论为基础开发了"SQL";
 - 1979: Oracle 发布了商业版SQL
 - 1983: db2版本面世
-
- SQL/86: ANSI 跟 ISO的第一个标准;
 - SQL/89: 增加了引用完整性(referential integrity);
 - SQL/92: 被数据库管理系统厂商广泛接受;
 - SQL/99: Core level跟其他8种相应的level, 包括递归查询, 程序跟流程控制;
 - SQL/2003: 包含了XML相关内容,自动生成列值(column values)
 - SQL/2006: 定义了SQL与XML(包含XQuery)的关联应用;

SQL语言组成

- ※ 数据定义语言(DDL), 例如: CREATE、DROP、ALTER等语句
- ※ 数据操作语言(DML), 例如: INSERT (插入)、UPDATE (修改)、DELETE (删除) 语句
- ※ 数据查询语言(DQL), 例如: SELECT语句
- ※ 数据控制语言(DCL), 例如: GRANT、REVOKE、COMMIT、ROLLBACK等语句

DDL语句



SQL语言—创建表

- 使用CREATE TABLE语句创建表，需指定表名、列名和数据类型。

```
CREATE TABLE PERS
( ID          SMALLINT          NOT NULL,
  NAME        VARCHAR(9),
  DEPT        SMALLINT WITH DEFAULT 10,
  JOB         CHAR(5),
  YEARS       SMALLINT,
  SALARY      DECIMAL(7,2),
  COMM        DECIMAL(7,2),
  BIRTH_DATE  DATE)
```

SQL语言—创建视图

- 使用**CREATE VIEW**语句创建视图，以限制对表的数据访问或代替常用的**SQL**查询语句。

下列语句创建 STAFF 表中部门 20 内的非经理人员的视图，其中工资和佣金不通过基表显示。

```
CREATE VIEW STAFF_ONLY  
AS SELECT ID, NAME, DEPT, JOB, YEARS  
FROM STAFF  
WHERE JOB <> 'Mgr' AND DEPT=20
```

在创建视图之后，下列语句显示视图的内容：

```
SELECT *  
FROM STAFF_ONLY
```

DML语句



SQL语言-插入数据

- `INSERT INTO table_name(col1) VALUES(a)`
- `INSERT INTO table_name(col1) VALUES(a),(b),(d)`
- `INSERT INTO table_name(col1) SELECT ...`

下列语句使用 `VALUES` 子句将一行数据插入 `PERS` 表中:

```
INSERT INTO PERS  
  VALUES (12, 'Harris', 20, 'Sales', 5, 18000, 1000, '1950-1-1')
```

SQL语言-插入数据

- 下列语句使用VALUES子句将三行插入表PERS中:

```
INSERT INTO PERS (NAME, JOB, ID)
VALUES ('Swagerman', 'Prgmr', 500),
       ('Limoges', 'Prgmr', 510),
       ('Li', 'Prgmr', 520)
```

下列示例从 STAFF 表中选择部门 38 的成員的数据，并将它插入 PERS 表中:

```
INSERT INTO PERS (ID, NAME, DEPT, JOB, YEARS, SALARY)
SELECT ID, NAME, DEPT, JOB, YEARS, SALARY
FROM STAFF
WHERE DEPT = 38
```

SQL语言-更新数据

- **UPDATE** table_name SET col1=val1,col2=val2,...
WHERE ...
- 使用**UPDATE**语句来更改表中的数据，可以更改满足**WHERE**子句搜索条件的每行中的一列或多列的值。

下列示例更新 ID 为 410 的雇员的信息:

```
UPDATE PERS  
  SET JOB='Prgmr', SALARY = SALARY + 300  
  WHERE ID = 410
```

SET 子句指定要更新的列并提供值。

WHERE 子句是可选的，它指定要更新的行。如果省略 WHERE 子句，则数据库管理程序用您提供的值更新表或视图中的每一行。

SQL语言-更新数据

UPDATE

(

SELECT * FROM STUDENT WHERE NAME='张三'

)

SET BIRTHDAY='1991-1-5'

SQL语言-更新数据

```
CREATE TABLE TRANSACTION
```

```
(
```

```
    CUSTOMERID VARCHAR(10), ---顾客号
```

```
    SEQ INT NOT NULL, ---流水号 (每个顾客从 1 开始)
```

```
    PROCESSDATE DATE, ---处理日
```

```
    AMOUNT DECIMAL(16,4) ---金额
```

```
);
```

```
UPDATE
```

```
(
```

```
    SELECT
```

```
        TT.*,
```

```
        ROW_NUMBER() OVER() AS RN
```

```
    FROM
```

```
        TRANSACTION AS TT WHERE CUSTOMERID=...
```

```
)
```

```
SET SEQ=RN
```


SQL语言-删除数据

- `DELETE FROM table_name WHERE ...`
- 使用**DELETE**语句，根据在**WHERE**子句中指定的搜索条件从表中删除数据行。

下列示例删除其中雇员 ID 为 120 的行:

```
DELETE FROM PERS  
WHERE ID = 120
```

WHERE 子句是可选的，它指定要删除的行。如果省略 WHERE 子句，则数据库管理程序删除表或视图中的所有行。



```
DELETE FROM
```

```
(
```

```
SELECT * FROM <TABLE_NAME> WHERE <CONDITION>
```

```
);
```

- 批量删除数据性能更好

清数据办法

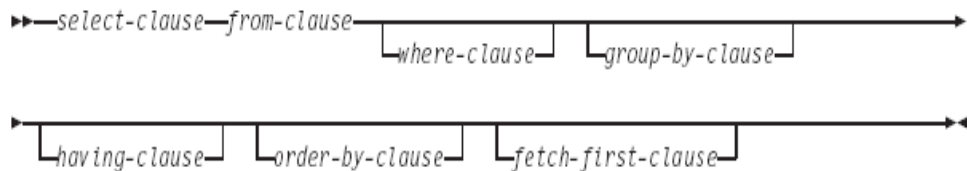
- Delete from ...
- Drop/create
- Alter table .. Activate not logged initially with empty table
- Load from /dev/null of del replace into ...
- Truncate

SELECT语句



查询语句

- Select基本语法:



查询语句-选择列

使用 SELECT 语句从表中选择特定的列。在该语句中指定用逗号分隔的列名列表。此列表称为选择列表。

下列语句从 SAMPLE 数据库的 ORG 表中选择部门名称 (DEPTNAME) 和部门号 (DEPTNUMB):

```
SELECT DEPTNAME, DEPTNUMB  
FROM ORG
```

通过使用星号 (*) 可从表中选择所有列。下一个示例列出了 ORG 表中的所有的列和行:

```
SELECT *  
FROM ORG
```

查询语句-选择行

要从表中选择特定行，在 SELECT 语句之后使用 WHERE 子句指定要选择的行必须满足的条件。从表中选择行的标准是搜索条件。

搜索条件由一个或多个谓词组成。谓词指定关于某一行是真或是假（或未知）的条件。可使用下列基本谓词在 WHERE 子句中指定条件：

谓词	功能
$x = y$	x 等于 y
$x \neq y$	x 不等于 y
$x < y$	x 小于 y
$x > y$	x 大于 y
$x \leq y$	x 小于或等于 y
$x \geq y$	x 大于或等于 y
IS NULL/IS NOT NULL	测试空值

查询语句-选择行

下列示例只从 STAFF 表中选择部门 20 的行:

```
SELECT DEPT, NAME, JOB  
FROM STAFF  
WHERE DEPT = 20
```

可以使用**AND**和**OR**来指定任意多个查询条件。

```
SELECT DEPT, NAME, JOB  
FROM STAFF  
WHERE JOB = 'Clerk'  
AND DEPT = 20
```

OR or IN

- 查找出生日期是'1949-10-1' 或'1978-12-18'的用户

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

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

查询语句-选择行

使用谓词 IS NULL 和 IS NOT NULL 来检查空值。

下列语句列出了佣金未知的雇员:

```
SELECT ID, NAME  
FROM STAFF  
WHERE COMM IS NULL
```

查询语句-排序

您可能想要信息按特定次序返回。使用 ORDER BY 子句将信息按一个或多个列中的值进行排序。

下列语句显示部门 84 中按雇用年数排序的雇员:

```
SELECT NAME, JOB, YEARS  
FROM STAFF  
WHERE DEPT = 84  
ORDER BY YEARS
```

指定 ORDER BY 作为整个 SELECT 语句中的最后一个子句。在此子句中命名的列可以是表达式或表的任何列。ORDER BY 子句中的列名不必在选择列表中指定。

可通过在 ORDER BY 子句中显式指定 ASC 或 DESC 将行按升序或降序进行排序。如果既未指定 ASC, 也未指定 DESC, 则自动按升序将行进行排序。

查询语句-去除重复行

当使用 SELECT 语句时，您可能不想要返回重复信息。例如，STAFF 有一个其中多次列出了几个部门号的 DEPT 列，以及一个其中多次列出了几个工作说明的 JOB 列。

要消除重复行，在 SELECT 子句上使用 DISTINCT 选项。例如，如果将 DISTINCT 插入该语句，则部门中的每项工作仅列出一次：

```
SELECT DISTINCT DEPT, JOB  
FROM STAFF  
WHERE DEPT < 30  
ORDER BY DEPT, JOB
```

查询语句-运算次序

- 查询语句的运算次序如下，一个子句的输出是下一个子句的输入。只有运算次序在后面的子句才能引用运算次序在前面的子句。

1. FROM 子句
2. WHERE 子句
3. GROUP BY 子句
4. HAVING 子句
5. SELECT 子句
6. ORDER BY 子句

查询语句-给表达式命名

可选的 AS 子句允许您给表达式指定有意义的名称，这就使得以后再引用该表达式更容易。可使用 AS 子句为选择列表中的任何项提供名称。

下列语句显示其工资加佣金少于 \$13,000 的所有雇员。表达式 SALARY + COMM 命名为 PAY:

```
SELECT NAME, JOB, SALARY + COMM AS PAY
FROM STAFF
WHERE (SALARY + COMM) < 13000
ORDER BY PAY
```

通过使用 AS 子句，可以在 ORDER BY 子句中引用特定的列名而不是系统生成的数字。在此示例中，我们在 WHERE 子句中将 (SALARY + COMM) 与 13000 进行比较，而不是使用名称 PAY。这是运算次序的结果。在给定 (SALARY + COMM) 名称 PAY 之前计算 WHERE 子句的值，原因是 SELECT 子句在 WHERE 子句后执行。因此，不能在该谓词中使用 PAY。

查询语句-分组

可根据 GROUP BY 子句中定义的分组结构来组织行。最简单的格式为，一个组就是一组行，每一组在“GROUP BY”列中都具有完全相同的值。SELECT 子句中的列名必须为分组列或列函数。列函数对 GROUP BY 子句定义的每个组返回一个值。每一组由结果集中的单一行表示。下列示例产生一个列出了每个部门号的最高工资的结果：

```
SELECT DEPT, MAX(SALARY) AS MAXIMUM  
FROM STAFF  
GROUP BY DEPT
```

将WHERE子句与GROUP BY子句一起使用

分组查询可以在形成组和计算列函数之前具有消除非限定行的标准 WHERE 子句。必须在 GROUP BY 子句之前指定 WHERE 子句。例如：

```
SELECT WORKDEPT, EDLEVEL, MAX(SALARY) AS MAXIMUM  
FROM EMPLOYEE  
WHERE HIREDATE > '1979-01-01'  
GROUP BY WORKDEPT, EDLEVEL  
ORDER BY WORKDEPT, EDLEVEL
```

注意：Select子句中的列名必须为分组列或列函数

查询语句-分组

- 在**GROUP BY**子句之后使用**HAVING**子句

可将限定条件应用于各个组，以便 DB2 仅对满足条件的组返回结果。为此，在 GROUP BY 子句后面包含一个 HAVING 子句。HAVING 子句可包含一个或多个用 AND 和 OR 连接的谓词。每个谓词将组特性（如 AVG(SALARY)）与下列之一进行比较：

- 该组的另一个特性

例如：

```
HAVING AVG(SALARY) > 2 * MIN(SALARY)
```

- 常数

例如：

```
HAVING AVG(SALARY) > 20000
```

例如，下列查询查找雇员数超过 4 的部门的最高和最低工资：

```
SELECT WORKDEPT, MAX(SALARY) AS MAXIMUM, MIN(SALARY) AS MINIMUM  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
HAVING COUNT(*) > 4  
ORDER BY WORKDEPT
```

查询语句-条件表达式

可在 SQL 语句中使用 CASE 表达式以便于处理表的数据表示。这提供了一种功能强大的条件表达式能力，在概念上与某些程序设计语言中的 CASE 语句类似。

- 要从 ORG 表中的 DEPTNAME 列将部门号更改为有意义的字，输入下列查询:


```
SELECT DEPTNAME,  
       CASE DEPTNUMB  
         WHEN 10 THEN 'Marketing'  
         WHEN 15 THEN 'Research'  
         WHEN 20 THEN 'Development'  
         WHEN 38 THEN 'Accounting'  
         ELSE 'Sales'  
       END AS FUNCTION  
FROM ORG
```

条件表达式例子



```
SELECT firstnme, lastname,  
CASE  
WHEN salary <= 40000 THEN 'Need a raise'  
WHEN salary > 40000 AND salary <= 50000  
    THEN 'Fair pay'  
ELSE 'Overpaid'  
END AS comment  
FROM employee
```

条件表达式



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

```
UPDATE USER SET BIRTHDAY=  
(  
CASE NAME  
WHEN '张三' THEN '1949-10-1'  
WHEN '李四' THEN '1997-7-1'  
ELSE BIRTHDAY  
END  
)  
where NAME in ('张三','李四');
```


查询语句-相关名

相关名是用于识别一个对象的多种用途的标识符。可在查询的 FROM 子句中和 UPDATE 或 DELETE 语句的第一个子句中定义相关名。相关名可与表、视图或嵌套表表达式关联，但只限于定义相关名的上下文中。

例如，子句 FROM STAFF S、ORG O 分别指定 S 和 O 作为 STAFF 和 ORG 的相关名。

```
SELECT NAME, DEPTNAME  
FROM STAFF S, ORG O  
WHERE O.MANAGER = S.ID
```

一旦定义了相关名，则只能使用相关名来限定对象。例如，在上例中，如果写成 ORG.MANAGER=STAFF.ID 的话，则该语句就会失效。

查询语句-集合运算

UNION、EXCEPT 以及 INTERSECT 集合运算符使您能够将两个或更多外层查询组合成单个查询。执行由这些集合运算符连接的每个查询，并将各个结果结合起来。每个运算符产生不同的结果。

UNION 运算符通过组合其他两个结果表（例如 TABLE1 和 TABLE2）并消去表中任何重复行而派生出一个结果表。当 ALL 随 UNION 一起使用时（即 UNION ALL），不消除重复行。两种情况下，派生表的每一行不是来自 TABLE1 就是来自 TABLE2。

在下列 UNION 运算符的示例中，查询返回工资高于 \$21,000、有管理责任且工龄少于 8 年的人员的姓名：

```
SELECT ID, NAME FROM STAFF WHERE SALARY > 21000  
UNION  
SELECT ID, NAME FROM STAFF WHERE JOB='Mgr' AND YEARS < 8  
ORDER BY ID
```

查询语句-集合运算

EXCEPT 运算符通过包括所有在 TABLE1 中但不在 TABLE2 中的行并消除所有重复行而派生出一个结果表。当 ALL 随 EXCEPT 一起使用时 (EXCEPT ALL), 不消除重复行。

在以下 EXCEPT 运算符的示例中, 查询返回收入超过 \$21,000, 但没有经理职位而工龄为 8 年或更长的所有人员的姓名。

```
SELECT ID, NAME FROM STAFF WHERE SALARY > 21000  
EXCEPT  
SELECT ID, NAME FROM STAFF WHERE JOB='Mgr' AND YEARS < 8
```

查询语句-集合运算

INTERSECT 运算符通过只包括 TABLE1 和 TABLE2 中都有的行并消除所有重复行而派生出一个结果表。当 ALL 随 INTERSECT 一起使用时 (INTERSECT ALL), 不消除重复行。

在以下 INTERSECT 运算符的示例中, 查询返回收入超过 \$21,000, 有管理职位且工龄少于 8 年的雇员的姓名和 ID。

```
SELECT ID, NAME FROM STAFF WHERE SALARY > 21000  
INTERSECT  
SELECT ID, NAME FROM STAFF WHERE JOB='Mgr' AND YEARS < 8
```

查询语句-集合运算

当使用 UNION、EXCEPT 以及 INTERSECT 运算符时，记住下列事项：

- 运算符的查询选择列表中的所有对应项必须是相容的。有关详情，参见 *SQL Reference* 中的数据类型相容性表。
- ORDER BY 子句（如果使用该子句的话）必须放在最后一个带有集合运算符的查询后面。对于每个运算符来说，仅当列名与查询的选择列表中的对应项完全相同时，该列名才能在 ORDER BY 子句中使用。
- 在具有相同数据类型和相同长度的列之间进行的运算会产生一个具有该类型和长度的列。针对 UNION、EXCEPT 以及 INTERSECT 集合运算符的结果，参见 *SQL Reference* 中结果数据类型的规则。

查询语句-IN谓词

使用 IN 谓词将一个值与其他几个值进行比较。例如:

```
SELECT NAME  
FROM STAFF  
WHERE DEPT IN (20, 15)
```

此示例相当于:

```
SELECT NAME  
FROM STAFF  
WHERE DEPT = 20 OR DEPT = 15
```

Tips: 在SQL语句中应该尽量避免使用OR，因为会影响SQL语句的性能。

查询语句-BETWEEN谓词

BETWEEN 谓词将单一值与一个范围内的值（在 BETWEEN 谓词中命名）作比较。

以下示例查找收入在 \$10,000 和 \$20,000 之间的雇员的姓名:

```
SELECT LASTNAME  
FROM EMPLOYEE  
WHERE SALARY BETWEEN 10000 AND 20000
```

这相当于:

```
SELECT LASTNAME  
FROM EMPLOYEE  
WHERE SALARY >= 10000 AND SALARY <= 20000
```

下一个示例查找收入少于 \$10,000 或超过 \$20,000 的雇员的姓名:

```
SELECT LASTNAME  
FROM EMPLOYEE  
WHERE SALARY NOT BETWEEN 10000 AND 20000
```

查询语句-LIKE谓词

使用 LIKE 谓词搜索具有某些模式的字符串。通过百分号和下划线指定模式。

- 下划线字符(_)表示任何单个字符。
- 百分号(%)表示零或多个字符的字符串。
- 任何其他表示本身的字符。

以下示例选择长为 7 个字母且以字母 'S' 开头的雇员姓名:

```
SELECT NAME  
FROM STAFF  
WHERE NAME LIKE 'S _ _ _ _ _ _ _'
```

下一个示例选择不以字母'S'开头的雇员名:

```
SELECT NAME  
FROM STAFF  
WHERE NAME NOT LIKE 'S%'
```


- **LIKE谓词过滤率计算公式：**DB2会将**LIKE**谓词作为**BETWEEN**谓词进行处理。谓词中界定范围的两个值产生于谓词中的字符串。只有特定字符（'%'或者'_'）前面的前导字符用于生成界定范围的值。所以如果特定字符是字符串中的第一个字符，那么过滤率被估计为1且这个谓词被估计不会过滤掉任何记录。

```
CREATE TABLE USER
```

```
(  
  NAME VARCHAR(20) NOT NULL, ---姓名  
  MYNUMBER VARCHAR(18) ---身份证号码  
);
```

问题：把身份证号码开头是2102（大连人）查出来

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

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

谓词类型	可索引	注 释
Col \in con	Y	\in 代表>, >=, =, <=, <, 但是<>是可能不可索引的。
Col between con1 and con2	Y	在匹配系列中必须是最后的。
Col in list	Y	仅对一个匹配列
Col is null	Y	
Col like 'xyz%'	Y	模糊匹配%在后面。
Col like '%xyz'	N	模糊匹配%在前面。
Col1 \in Col2	N	Col1和col2来自同一个表
Col \in Expression	N	例如: c1 (c1+1) /2
Pred1 and Pred2	Y	Pred1和Pred2都是可索引的, 指相同索引的列
Pred1 or Pred2	N	除了 (c1=a or c1=b) 外, 他可以被认为是c1 in (a, b)
Not Pred1	N	或者任何的等价形式: Not between, Not in, Not like等等。

思考题一：

请编写SQL语句，找出身份证相同的记录

举例：

```
create table testtab (idno char(15), balance int)
```

数据：

idno	balance
222328192232122	19999
422114346456556	2323212
222328192232122	32232
111111111111111	43444
222328192232122	344554
111111111111111	3222

其中一种写法：

```
select * from testtab where idno in (select idno from  
testtab group by idno having count(idno)>1 )
```



连接查询

查询语句-连接

从两个或更多个表中组合数据的过程称为连接表。数据库管理程序从指定的表中形成行的所有组合。对于每个组合，它都测试连接条件。连接条件是带有一些约束的搜索条件。有关约束的列表，参考 *SQL Reference*。

注意，连接条件涉及的列的数据类型不必相同；然而，这些数据类型必须相容。计算连接条件的方式与计算其他搜索条件的方式相同，并且使用相同的比较规则。

如果未指定连接条件，则返回在 FROM 子句中列出的表中行的所有组合，即使这些行可能完全不相关。该结果称为这两个表的交叉积。

以下示例产生两个表的交叉积。未指定连接条件，因而存在每一个行组合：

```
SELECT SAMP_PROJECT.NAME,  
       SAMP_PROJECT.PROJ, SAMP_STAFF.NAME, SAMP_STAFF.JOB  
FROM SAMP_PROJECT, SAMP_STAFF
```

查询语句-连接

两个主要的连接类型是内连接和外连接。到目前为止，所有示例中使用的都是内连接。内连接只保留交叉积中满足连接条件的那些行。如果某行在一个表中存在，但在另一个表中不存在，则结果表中不包括该信息。

下列示例产生两个表的内连接。该内连接列出分配给某个项目的全职雇员：

```
SELECT SAMP_PROJECT.NAME,  
        SAMP_PROJECT.PROJ, SAMP_STAFF.NAME, SAMP_STAFF.JOB  
FROM SAMP_PROJECT, SAMP_STAFF  
WHERE SAMP_STAFF.NAME = SAMP_PROJECT.NAME
```

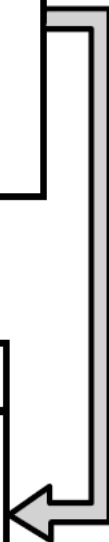
从多表中获取数据（原理）

PROJECT

PROJNO	PROJNAME	DEPTNO	...
AD3100	ADMIN SERVICES	D01	
AD3110	GENERAL ADMIN SYSTEMS	D21	
AD3111	PAYROLL PROGRAMMING	D21	
AD3112	PERSONNEL PROGRAMMING	D21	
:	:		

DEPARTMENT

DEPTNO	DEPTNAME	...
A00	SPIFFY COMPUTER SERVICE DIV.	
C01	INFORMATION CENTER	
D01	DEVELOPMENT CENTER	
D21	ADMINISTRATION SYSTEMS	
:	:	



从多表获取数据(Join)

For every project, list the project number, project name, and the number and name of the department responsible for the project.

```
SELECT      PROJNO, PROJNAME, PROJECT.DEPTNO, DEPTNAME
FROM        PROJECT, DEPARTMENT
WHERE       PROJECT.DEPTNO=DEPARTMENT.DEPTNO -- JOIN PREDICATE
ORDER BY    PROJNO
```



<u>PROJNO</u>	<u>PROJNAME</u>	<u>DEPTNO</u>	<u>DEPTNAME</u>
AD3100	ADMIN SERVICES	D01	DEVELOPMENT CENTER
AD3110	GENERAL ADMIN SYSTEMS	D21	ADMINISTRATION SYSTEMS
AD3111	PAYROLL PROGRAMMING	D21	ADMINISTRATION SYSTEMS
AD3112	PERSONNEL PROGRAMMING	D21	ADMINISTRATION SYSTEMS
AD3113	ACCOUNT PROGRAMMING	D21	ADMINISTRATION SYSTEMS
...

Avoid a Cartesian Product!

三个表做Join (1)

PROJECT			
PROJNO	PROJNAME	DEPTNO	...
AD3100	ADMIN SERVICES	D01	
AD3110	GENERAL AD SYSTEMS	D21	
AD3111	PAYROLL PROGRAMMING	D21	
AD3112	PERSONNEL PROGRAMMING	D21	
AD3113	ACCOUNT. PROGRAMMING	D21	
IF1000	QUERY SERVICES	C01	
.	.	.	.

For department D21 list
PROJNO, DEPTNO,
DEPTNAME, MGRNO, and
LASTNAME.

DEPARTMENT			
DEPTNO	DEPTNAME	MGRNO	...
A00	SPIFFY COMPUTER SERVICE DIV	000010	
B01	PLANNING	000020	
C01	INFORMATION CENTER	000030	
D01	DEVELOPMENT CENTER	-----	
D11	MANUFACTURING SYSTEMS	000060	
D21	ADMINISTRATION SYSTEMS	000070	
E01	SUPPORT SERVICES	000050	
.	.	.	.

EMPLOYEE				
EMPNO	FIRSTNAME	MIDINIT	LASTNAME	...
000010	CHRISTINE	I	HAAS	
000020	MICHAEL	L	THOMPSON	
000030	SALLY	A	KWAN	
000050	JOHN	B	GEYER	
000060	IRVING	F	STERN	
000070	EVA	D	PULASKI	
000090	EILEEN	W	HENDERSON	
000100	THEODORE	Q	SPENSER	
.

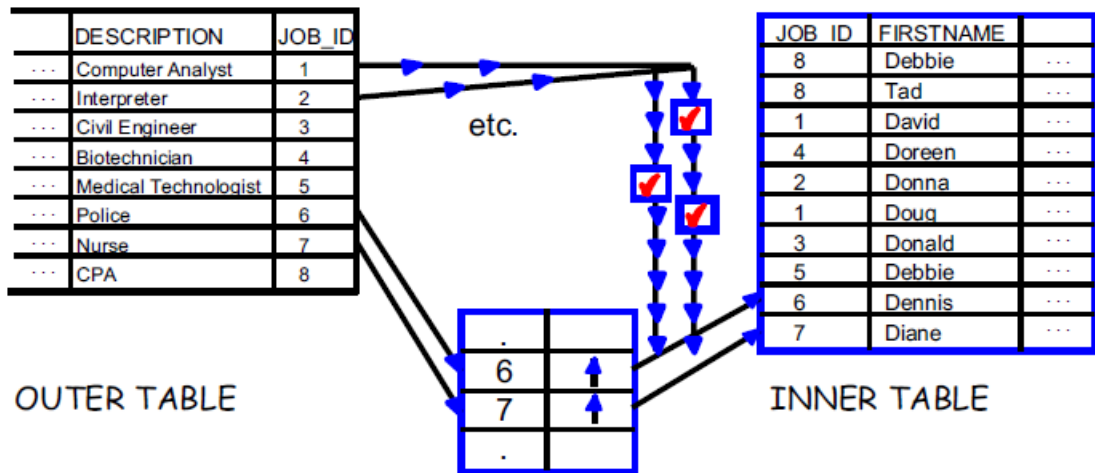
三个表做Join (2)

```
SELECT      PROJNO, P.DEPTNO, DEPTNAME, MGRNO, LASTNAME
FROM        PROJECT P,
            DEPARTMENT D,
            EMPLOYEE E
WHERE       P.DEPTNO = D.DEPTNO      -- join predicate
           AND D.MGRNO = E.EMPNO     -- join predicate
           AND D.DEPTNO = 'D21'     -- local predicate
ORDER BY    PROJNO
```



PROJNO	DEPTNO	DEPTNAME	MGRNO	LASTNAME
AD3110	D21	ADMINISTRATION SYSTEMS	000070	PULASKI
AD3111	D21	ADMINISTRATION SYSTEMS	000070	PULASKI
AD3112	D21	ADMINISTRATION SYSTEMS	000070	PULASKI
AD3113	D21	ADMINISTRATION SYSTEMS	000070	PULASKI

Join 方法 (Nested-loop join)



```
SELECT *  
FROM SIBLINGS S,  
OCCUPATIONS O  
WHERE  
S.JOB_ID = O.JOB_ID
```

```

794.52
TBSCAN
( 2)
3827.09
317.376
|
794.52
SORT
( 3)
3827.02
317.376
|
794.52
NLJOIN
( 4)
3826.48
317.376

```

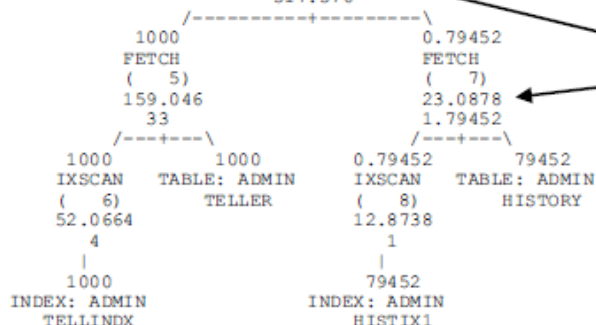
```

SELECT HISTORY.BRANCH_ID,
TELLER.TELLER NAME,
HISTORY.ACCTNAME,
HISTORY.ACCT_ID, HISTORY.BALANCE
FROM HISTORY AS HISTORY, TELLER AS TELLER
WHERE HISTORY.TELLER_ID =
TELLER.TELLER_ID AND HISTORY.BRANCH_ID = 25
ORDER BY HISTORY.BRANCH_ID ASC,
HISTORY.ACCT_ID ASC ;

```

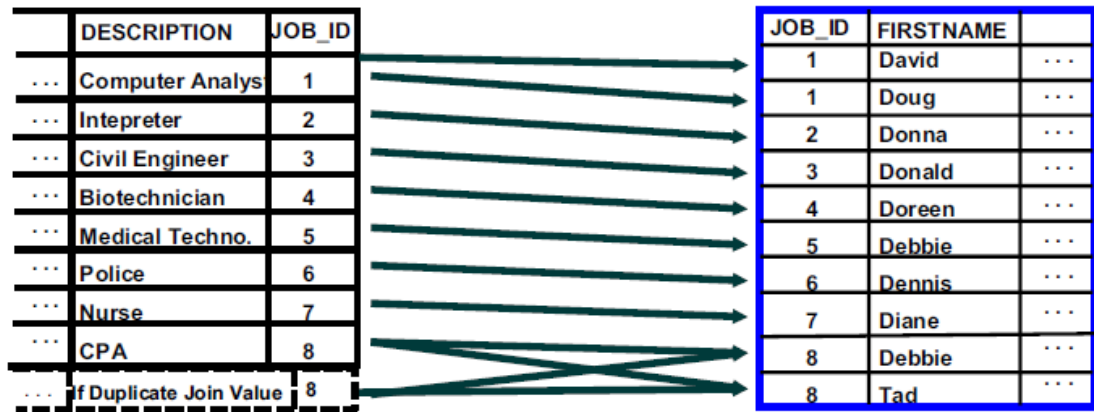
OUTER Table

INNER Table



- Inner Table access cost is based on matching one outer table row.
- Join Cost is adjusted based on index clustering

Join 方法（Merge join）



```
SELECT  *
FROM    SIBLINGS  S,
OCCUPATIONS  O
WHERE
S.JOB_ID = O.JOB_ID
```

- JOIN SATISFIED THROUGH TABLES ORDERED ON JOIN COLUMN(S), INDEXES COULD REDUCE SORT COSTS.

Index Used for
Ordering
By Join Column

```

794.52
MSJOIN
( 4)
830.571
375
/-----+-----\
1000 0.79452
FETCH FILTER
( 5) ( 7)
159.046 671.23
33 342
/-----+-----\
1000 1000 794.52
IXSCAN TABLE: ADMIN TBSCAN
( 6) TELLER ( 8)
52.0664 671.23
4 342
|
1000 794.52
INDEX: ADMIN SORT
TELLINDX ( 9)
671.229
342
|
794.52
TBSCAN
( 10)
670.873
342
|
79452
TABLE: ADMIN
HISTORY

```

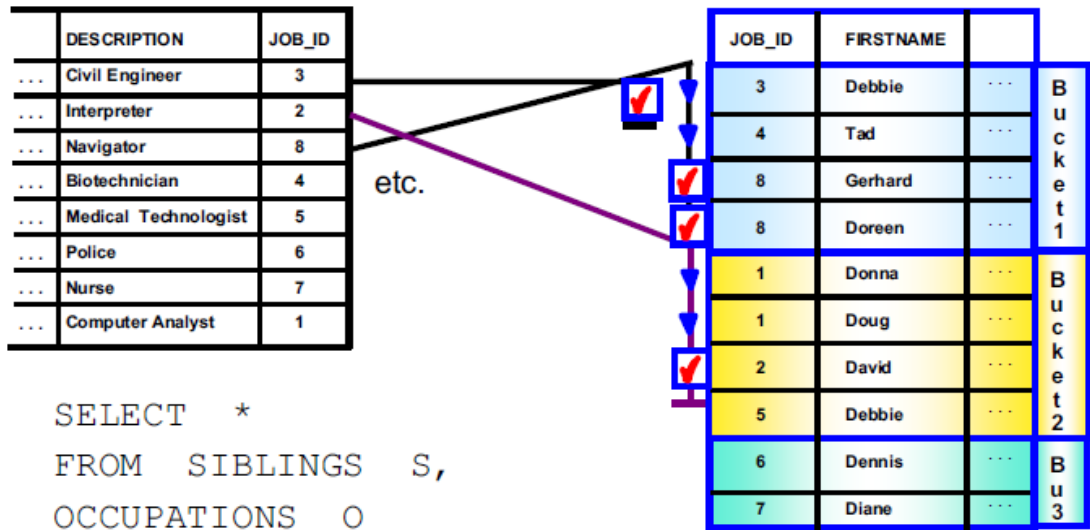
SELECT HISTORY.BRANCH_ID,
TELLER.TELLER_NAME,
HISTORY.ACCTNAME,
HISTORY.ACCT_ID, HISTORY.BALANCE
FROM HISTORY AS HISTORY, TELLER AS TELLER
WHERE HISTORY.TELLER_ID = TELLER.TELLER_ID
AND HISTORY.BRANCH_ID = 25
ORDER BY HISTORY.BRANCH_ID ASC,
HISTORY.ACCT_ID ASC ;

Sort Used for Ordering
By Join Column

OUTER Table

INNER Table

Join 方法 (Hash join)



```
SELECT  *
FROM    SIBLINGS  S,
        OCCUPATIONS  O
WHERE   S.JOB_ID = O.JOB_ID
```

OUTER Table

```

794.52
HSJOIN
( 4)
722.496
336.17

/-----+-----\
794.52          1000
FETCH          FETCH
( 5)          ( 9)
563.302        159.046
303.17         33
/-----+-----\

794.52          79452
RIDSCN        TABLE: ADMIN
ADMIN
( 6)          HISTORY
13.6858
1
|
794.52
SORT
( 7)
13.6853
1
|
794.52
IXSCAN
( 8)
13.3792
1
|
79452
INDEX: ADMIN
HISTIX1

```

INNER Table

```

1000
FETCH
( 9)
159.046
33
/-----+-----\

1000          1000
IXSCAN        TABLE:
TELLER
( 10)
52.0664
4
|
1000
INDEX: ADMIN
TELLINDX

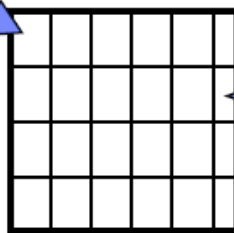
```

```


SELECT HISTORY.BRANCH_ID,
TELLER.TELLER_NAME,
HISTORY.ACCTNAME,
HISTORY.ACCT_ID, HISTORY.BALANCE
FROM HISTORY AS HISTORY, TELLER AS TELLER
WHERE HISTORY.TELLER_ID =
TELLER.TELLER_ID AND HISTORY.BRANCH_ID = 25
ORDER BY HISTORY.BRANCH_ID ASC,
HISTORY.ACCT_ID ASC ;

```

Sortheap



查询语句-连接

- 
1. 左外连接包括内连接和左表中未包括在内连接中的那些行。
 2. 右外连接包括内连接和右表中未包括在内连接中的那些行。
 3. 全外连接包括内连接以及左表和右表中未包括在内连接中的行。

使用 SELECT 语句来指定要显示的列。在 FROM 子句中，列出后跟关键字 LEFT OUTER JOIN、RIGHT OUTER JOIN 或 FULL OUTER JOIN 的第一个表的名称。接着需要指定后跟 ON 关键字的第二个表。在 ON 关键字后面，指定表示要连接的表之间关系的连接条件。

左外连接(left outer join)

- 左外连接：以左表为基础来连接,如果左表的某行内容无法在右表中找到相对的row,则将右表统统用null来表示

在下列示例中，将 SAMP_STAFF 指定为右表，而 SAMP_PROJECT 则被指定为左表。通过使用 LEFT OUTER JOIN，列出所有全职雇员和合同雇员（在 SAMP_PROJECT 中列出）的姓名和项目号，如果是全职雇员（在 SAMP_STAFF 中列出），还列出这些雇员的职位：

```
SELECT SAMP_PROJECT.NAME, SAMP_PROJECT.PROJ,  
       SAMP_STAFF.NAME, SAMP_STAFF.JOB  
FROM SAMP_PROJECT LEFT OUTER JOIN SAMP_STAFF  
     ON SAMP_STAFF.NAME = SAMP_PROJECT.NAME
```

此语句产生下列结果：

NAME	PROJ	NAME	JOB
Haas	AD3100	Haas	PRES
Lutz	MA2111	-	-
Thompson	PL2100	Thompson	MANAGER
Walker	MA2112	-	-

右外连接(right outer join)

- 右外连接：以右表为基础来连接,如果右表的某行内容无法在左表中找到相对的row,则将左表统统用null来表示

在下一个示例中，将 SAMP_STAFF 指定为右表而 SAMP_PROJECT 则被指定为左表。通过使用 RIGHT OUTER JOIN 列出所有全职雇员（在 SAMP_STAFF 中列出）的姓名和工作职位，如果将这些雇员分配给了某个项目（在 SAMP_PROJECT 中列出），还列出他们的项目编号：

```
SELECT SAMP_PROJECT.NAME,  
       SAMP_PROJECT.PROJ, SAMP_STAFF.NAME, SAMP_STAFF.JOB  
FROM SAMP_PROJECT RIGHT OUTER JOIN SAMP_STAFF  
     ON SAMP_STAFF.NAME = SAMP_PROJECT.NAME
```

NAME	PROJ	NAME	JOB
-----		-----	
Haas	AD3100	Haas	PRES
-	-	Lucchessi	SALESREP
-	-	Nicholls	ANALYST
Thompson	PL2100	Thompson	MANAGER

思考题一：

- 大表关联出现NL-Join或Merge-Join时要特别小心
- 当NL或Merge Join的结果 <1 时要特别小心



子查询

查询语句-子查询

--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
);
```

- 问题：
查找公司电话是88888888的用户有哪些？

- 非相关子查询(独立查询):

--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

--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.COMPANYID
);
```

独立查询的例子

Whose salary is higher
than the average salary?



First Select:

```
SELECT  
FROM
```

```
AVG(SALARY)  
EMPLOYEE
```



27303

Second Select:

```
SELECT  
FROM  
WHERE
```

```
EMPNO, LASTNAME  
EMPLOYEE  
SALARY > 27303
```

独立查询的例子

- 标量全查询返回一行，且该行只有一个值。
- 以下示例列出了工资超过全部雇员平均工资的雇员的姓名。查询中的标量全查询是用括号括起来的选择语句。

```
SELECT LASTNAME, FIRSTNAME  
FROM EMPLOYEE  
WHERE SALARY > (SELECT AVG(SALARY)  
FROM EMPLOYEE)
```


查询语句-嵌套表表达式

嵌套表表达式是一个临时视图，其中的定义被嵌套（直接定义）在主查询的 FROM 子句中。

下列查询使用嵌套表表达式来查找那些教育级别超过 16 的雇员的平均总收入、教育级别以及雇用年份：

```
SELECT EDLEVEL, HIREYEAR, DECIMAL(AVG(TOTAL_PAY),7,2)
  FROM (SELECT EDLEVEL, YEAR(HIREDATE) AS HIREYEAR,
              SALARY+BONUS+COMM AS TOTAL_PAY
        FROM EMPLOYEE
        WHERE EDLEVEL > 16) AS PAY_LEVEL
 GROUP BY EDLEVEL, HIREYEAR
 ORDER BY EDLEVEL, HIREYEAR
```

查询语句-相关子查询

允许引用先前提到的任何表的子查询称为*相关子查询*。我们也说该子查询具有对主查询中表的*相关引用*。

以下示例使用相关子查询来列出其工资高于部门平均工资的所有雇员：

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT
FROM EMPLOYEE E1
WHERE SALARY > (SELECT AVG(SALARY)
                  FROM EMPLOYEE E2
                  WHERE E2.WORKDEPT = E1.WORKDEPT)
ORDER BY E1.WORKDEPT
```

相关子查询

--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
);
```

- 非相关子查询(独立查询):

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

- 相关子查询

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

- 问题:
如何把用户电话更新成公司电话?

使用IN的嵌套子查询

List the names and employee numbers of employees who are managers of a department



```
SELECT FIRSTNAME, LASTNAME, EMPNO  
FROM EMPLOYEE  
WHERE EMPNO IN (SELECT MGRNO  
FROM DEPARTMENT)
```

Final result

FIRSTNAME	LASTNAME	EMPNO
CHRISTINE	HAAS	000010
MICHAEL	THOMPSON	000020
SALLY	KWAN	000030
JOHN	GEYER	000050
IRVING	STERN	000060
EVA	PULASKI	000070
EILEEN	HENDERSE N	000090
THEODORE	SPENSER	000100

Result of subquery

000010
000020
000030
000050
000060
000070
000090
000100
-

嵌套查询（子查询）

- 带有ANY或ALL谓词的子查询 **some=any**

- **ANY**: 任意一个值

- **ALL**: 所有值

- 配合使用比较运算符的含义

- **> ANY >MIN**

大于子查询结果中的某个值

- **> ALL >MAX**

大于子查询结果中的所有值

- **< ANY <MAX**

小于子查询结果中的某个值

- **< ALL <MIN**

小于子查询结果中的所有值

- **>= ANY**

大于等于子查询结果中的某个值

- **>= ALL**

大于等于子查询结果中的所有值

- **<= ANY**

小于等于子查询结果中的某个值

- **<= ALL**

小于等于子查询结果中的所有值

- **= ANY IN**

等于子查询结果中的某个值

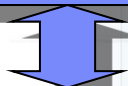


嵌套查询（子查询）

- 带有ANY或ALL谓词的子查询

例：查询比30部门工资高的员工的编号，姓名和工资。

```
SELECT empno, ename, sal
FROM emp
WHERE sal > ANY (SELECT sal FROM emp
                  WHERE deptno=30);
```



```
SELECT empno, ename, sal
FROM emp
WHERE sal > (SELECT min(sal) FROM emp
              WHERE deptno=30);
```

嵌套查询（子查询）

- 练习：查询比30部门所有员工工资都高的员工的姓名及工资

```
SELECT ename, sal
FROM emp
WHERE sal > ALL(SELECT sal FROM emp
                 WHERE deptno=30);
```

```
SELECT ename, sal
FROM emp
WHERE sal > (SELECT max(sal) FROM emp
             WHERE deptno=30);
```

多行子查询

使用 EXISTS 和 NOT EXISTS

- EXISTS的特征：
 - 使用 **EXISTS** 关键字引入一个子查询时，就相当于进行一次存在测试
 - 外部查询的 **WHERE** 子句测试子查询返回的行是否存在
 - 子查询返回的数据不受限制，**EXISTS**只考虑其结果集的行数是否为0
 - **EXISTS**一般用于子查询
- **NOT EXISTS**与**EXISTS**正相反



多行子查询

使用 **EXISTS** 和 **NOT EXISTS**

- 查询存在员工的部门信息:

```
select * from dept
```

```
where exists (select 1 from emp where emp.deptno = dept.deptno);
```

	DEPTNO	DNAME	LOC
1	10	ACCOUNTING	NEW YORK
2	20	RESEARCH	DALLAS
3	30	SALES	CHICAGO

多行子查询

使用 EXISTS 和 NOT EXISTS

- 查询不存在员工的部门信息：

```
select * from dept  
where not exists (select 1 from emp where emp.deptno = dept.deptno);
```

	DEPTNO	DNAME	LOC
1	40	OPERATIONS	BOSTON



IN与exists比较

- 如果两个表中一个较小，一个较大，则子查询表大的用**exists**，子查询表小的用**in**，因为**in**是把外表和内表作**hash**连接，而**exists**是对外表作**loop**循环，每次**loop**循环再对内表进行查询。
- 无论哪个表大，用**not exists**都比**not in**要快。这是因为如果查询语句使用了**not in**，那么内外表都进行全表扫描，没有用到索引；而**not exists**的子查询依然能用到表上的索引。
- **IN**的好处是逻辑直观简单（通常是独立子查询）；缺点是只能判断单字段，并且当**NOT IN**时效率较低，而且**NULL**会导致不想要的结果。
- **EXISTS**的好处是效率高，可以判断单字段和组合字段，并不受**NULL**的影响；缺点是逻辑稍微复杂（通常是相关子查询）。

子查询总结

- 相关子查询
 - 子查询语句依赖外部语句，不能单独执行
- 非相关子查询
 - 子查询语句可单独执行

--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  
);
```

问题1: 查询一下公司电话是**88888888** 的用户有哪些?

--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=COMPANYID  
);
```

- 
- 问题2：如何把用户电话更新成公司电话？

```
UPDATE USER AS U SET TELNO=
```

```
(
```

```
SELECT TELNO FROM COMPANY AS C WHERE U.COMPANYID=C.COMPANYID
```

```
);
```

临时表




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。

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;
```

DGTT=Declare Global Temporary Table



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;
```


- 
- DGTT=Declare Global Temporary Table
 - CGTT=Create Global Temporary Table


DGTT 与 CGTT 之间的主要差异在于 CGTT 的定义是持久存储在 DB2 catalog 的。DGTT 在创建后仅在用户会话期间保存。与 DGTT 不同，CGTT 在会话中创建并且在会话终止后仍然将持久保存。CGTT 定义由所有并发会话共享，即使其内容是各会话所私有的。

通过 CGTT 和 DGTT，应用程序会话可以使用已创建的临时表来保存操作或重复引用的中间结果集，而不会干扰并发运行的应用程序。

创建 CGTT 主要有两个动机。首先，CGTT 的行为对 SQL 程序员而言更像是一个普通表，但它具有提升性能的潜力。可以提前定义 CGTT 表，并且用户可以像普通表一样使用它。由于数据仅供各自会话使用（因此不需要行锁定），并且没有日志选项，因此提供了潜在的性能优势。

提供 CGTT 的第二个动机是帮助将非 DB2 临时表转换为 DB2 临时表。它可以降低这类操作的成本，从而能够更加轻松地迁移到 DB2。

CGTT定义



```
Create GLOBAL TEMPORARY TABLE <table_name>
( <column_name> <column_datatype>, <column_name> <column_datatype>, ... )
ON COMMIT [PRESERVE|DELETE] ROWS
[NOT LOGGED|LOGGED]
ON ROLLBACK [PRESERVE|DELETE] ROWS
DISTRIBUTE BY HASH ( col1,..)
IN <tspace-name>;
```

Merge



在更新数据的同时查看数据

```
SELECT * FROM FINAL TABLE
```

```
(
```

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

```
)
```

```
SELECT * FROM FINAL TABLE
```

```
(
```

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

```
)
```

```
WHERE NAME LIKE '李%'
```

```
FETCH FIRST 10 ROWS ONLY;
```

```
SELECT * FROM OLD TABLE
```

```
(
```

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

```
)
```

```
WHERE NAME LIKE '李%'
```

```
FETCH FIRST 10 ROWS ONLY;
```

Merge

- 将一个表的数据合并到另一个表中，合并的同时可以进行增删改。

---雇员表 (EMPLOYEE)

```
CREATE TABLE EMPLOYEE (  
  EMPLOYEEID INTEGER NOT NULL, ---员工号  
  NAME VARCHAR(20) NOT NULL, ---姓名  
  SALARY DOUBLE ---薪水  
);  
  
INSERT INTO EMPLOYEE (EMPLOYEEID, 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 (  
  EMPLOYEEID 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);
```

问题1：要求将Manager表的数据合并到Employee

```
MERGE INTO EMPLOYEE AS EM
```

```
  USING MANAGER AS MA
```

```
 ON EM.EMPLOYEEID=MA.MANAGERID
```

```
 WHEN MATCHED THEN UPDATE SET EM.SALARY=MA.SALARY
```

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



问题2: 如果经理表的薪水>雇员表的薪水的时候更新, 否则不更新

```
MERGE INTO EMPLOYE AS EM
USING MANAGER AS MA
ON EM.EMPLOYEEID=MA.MANAGERID
WHEN MATCHED AND EM.SALARY<MA.SALARY THEN UPDATE SET EM.SALARY=MA.SALARY
WHEN NOT MATCHED THEN INSERT VALUES (MA.MANAGERID,MA.NAME,MA.SALARY);
```

问题3: 抛异常

```
MERGE INTO EMPLOYE AS EM
USING MANAGER AS MA
ON EM.EMPLOYEEID=MA.MANAGERID
WHEN MATCHED AND EM.SALARY<MA.SALARY THEN UPDATE SET EM.SALARY=MA.SALARY
WHEN MATCHED AND EM.SALARY>MA.SALARY THEN SIGNAL SQLSTATE '70001' SET
MESSAGE_TEXT = 'EM.SALARY>MA.SALARY'
WHEN NOT MATCHED THEN INSERT VALUES (MA.MANAGERID,MA.NAME,MA.SALARY)
ELSE IGNORE;
```

采集样本数据

- 查看表中前10行数据：

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

- 随机查看表中前10行数据：

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

- 随机抽样：

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

Tips(1)

- 尽量避免在SQL语句的WHERE子句中使用函数

```
CREATE TABLE USER  
(  
  NAME VARCHAR(20) NOT NULL, ---姓名  
  REGISTERDATE TIMESTAMP ---注册时间  
);
```

问题1: 把2009.9.24 注册的用户都查出来

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



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

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


Tips(2)

- 尽量避免在**SQL**语句的**WHERE**子句中使用函数

```
CREATE TABLE USER  
(  
  NAME VARCHAR(20) NOT NULL, ---姓名  
  REGISTERDATE TIMESTAMP ---注册时间  
);
```

问题1: 把2009.9.24 注册的用户都查出来

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



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

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





DB2内置函数

- 
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

类型转换函数

- 类型转换函数

SMALLINT 返回 SMALLINT 类型的值

INTEGER 返回 INTEGER 类型的值

BIGINT 返回 BIGINT 类型的值

DECIMAL 返回 DECIMAL 类型的值

REAL 返回 REAL 类型的值

DOUBLE 返回 DOUBLE 类型的值

FLOAT 返回 FLOAT 类型的值

CHAR 返回 CHARACTER 类型的值

VARCHAR 返回 VARCHAR 类型的值

VARCHAR_FORMAT_BIT 将位字符序列格式化为 VARCHAR 类型返回

VARCHAR_BIT_FORMAT 将格式化后位字符序列返回到格式化前

SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE

如果想要一位小数，则转换成：

decimal(9,1)

字符串函数

ASCII 将字符转化为 ASCII 码

CHR 将 ASCII 码转化为字符

STRIP 删除字符串开始和结尾的空白字符或其他指定的字符

TRIM 删除字符串开始和结尾的空白字符或其他指定的字符

LTRIM 删除字符串开始的空白字符

RTRIM 删除字符串尾部的空白字符

LCASE or LOWER 返回字符串的小写

UCASE OR UPPER 返回字符串的大写

SUBSTR 返回子串

SUBSTRING 返回子串

LEFT 返回开始的 N 个字符

RIGHT 返回结尾的 N 个字符

POSITION 返回参数 2 在参数 1 中的第一次出现的位置

POSSTR 返回参数 2 在参数 1 中的第一次出现的位置

LOCATE 返回参数 2 在参数 1 中的第一次出现的位置

SPACE 返回由参数指定的长度, 包含空格在内的字符串

REPEAT 回参数 1 重复参数 2 次后的字符串

CONCAT 连接两个字符串

INSERT 向指定字符串添加字符串

REPLACE 替换字符串

TRANSLATE 将字符串中的一个或多个字符替换为其他字符

CHARACTER_LENGTH 返回字符串的长度

OCTET_LENGTH 返回字符串的字节数

ENCRYPT 对字符串加密

DECRYPT_BIN and DECRYPT_CHARS 对加密后的数据解密

GETHINT 返回密码提示

GENERATE_UNIQUE 生成唯一字符序列

日期函数

YEAR	返回日期的年部分
MONTH	返回日期的月部分
DAY	返回日期的日部分
HOURL	返回日期的小时部分
MINUTE	返回日期的分钟部分
SECOND	返回日期的秒部分
MICROSECOND	返回日期的微秒部分
MONTHNAME	返回日期的月份名称
DAYNAME	返回日期的星期名称
QUARTER	返回指定日期是第几季度
WEEK	返回当前日期是一年的第几周，每周从星期日开始
WEEK_ISO	返回当前日期是一年的第几周，每周从星期一开始
DAYOFWEEK	返回当前日期是一周的第几天，星期日是 1
DAYOFWEEK_ISO	返回当前日期是一周的第几天，星期一是 1
DAYOFYEAR	返回当前日期是一年的第几天
DAYS	返回用整数表示的时间，用来求时间间隔
JULIAN_DAY	返回从 January 1, 4712 B.C(Julian date calendar) 到指定日期的天数
MIDNIGHT_SECONDS	返回午夜到指定时间的秒数
TIMESTAMPDIFF	返回两个 timestamp 型日期的时间间隔
TIMESTAMP_ISO	返回 timestamp 类型的日期
TO_CHAR	返回日期的字符串表示
VARCHAR_FORMAT	将日期格式化为字符串
TO_DATE	将字符串转化为日期
TIMESTAMP_FORMAT	将字符串格式化为日期

Timestampdiff函数

- 求两个日期的时间差

```
select timestampdiff( 8, char(timestamp('2014-09-20-  
15.01.33.453312') - current timestamp))  
from sysibm.sysdummy1;
```

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。

参考链接：

<http://www.ibm.com/developerworks/data/library/techarticle/0211yip/0211yip3.html>

NULL值处理

COALESCE 将 **null** 转化为其他值

VALUE 将 **null** 转化为其他值

```
select coalesce(id,0) from <table_name>
```

create table user(name varchar(20) not null, salary float, bonus float)

问题：查找总工资(基本工资+奖金)大于20000 元的员工

```
select * from user where salary+bonus > 20000
```

缺陷：忽略null值

```
SELECT NAME FROM USER WHERE  
COALESCE(SALARY,0)+COALESCE(BONUS,0)>20000
```

函数分类-列函数

- 列函数对列中的一组值进行运算以得到单个结果值。

AVG 返回某一组中的值除以该组中值的个数的和

COUNT 返回一组行或值中行或值的个数

MAX 返回一组值中的最大值

MIN 返回一组值中的最小值

下列语句从 STAFF 表中选择最高工资:

```
SELECT MAX(SALARY)
FROM STAFF
```


函数分类-标量函数

- 标量函数对一个单一值进行某个运算以返回另一个单一值。

ABS	返回数的绝对值
HEX	返回值的十六进制表示
LENGTH	返回自变量中的字节数（对于图形字符串则返回双字节字符数。）
YEAR	抽取日期时间值的年份部分

下列语句返回 ORG 表中的部门名称以及其每个名称的长度:

```
SELECT DEPTNAME, LENGTH(DEPTNAME)
FROM ORG
```



With子句

WITH 语句的使用

With语句叫做CTE(common table expression)，其实就是定义临时集合。用查询(select)定义临时集合

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

With举例

```
CREATE TABLE USER
```

```
(
```

```
  NAME VARCHAR(20) NOT NULL,--姓名
```

```
  DEGREE INTEGER NOT NULL,--学历(1、专科2、本科3、硕士4、博士)
```

```
  STARTWORKDATE date NOT NULL,--入职时间
```

```
  SALARY1 FLOAT NOT NULL,--基本工资
```

```
  SALARY2 FLOAT NOT NULL--奖金
```

```
);
```

问题：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


查询学历是硕士或博士的那些员工得到结果集1：

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

根据学历和入职年份分组，求平均工资得到结果集2：

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

With例子实现(1)




```
WITH TEMP1(NAME,DEGREE,WORDDATE,SALARY) AS
(
  SELECT NAME,DEGREE,YEAR(STARTWORKDATE) AS
  WORDDATE, SALARY1+SALARY2 AS  SALARY FROM USER
  WHERE DEGREE IN (3,4)
),
TEMP2 (DEGREE,WORDDATE,AVG_SALARY) AS
(
  SELECT DEGREE,YEAR(STARTWORKDATE) AS WORDDATE,
  AVG(SALARY1+SALARY2) AS AVG_SALARY FROM USER WHERE
  DEGREE IN (3,4)
  GROUP BY DEGREE,YEAR(STARTWORKDATE)
)
SELECT NAME FROM TEMP1, TEMP2 WHERE
TEMP1.DEGREE=TEMP2.DEGREE
AND TEMP1.WORDDATE=TEMP2.WORDDATE
AND SALARY<AVG_SALARY;
```

With例子实现(2) – 改进

```
WITH TEMP1(NAME,DEGREE,WORDDATE,SALARY) AS
(
  SELECT NAME,DEGREE,YEAR(STARTWORKDATE) AS
  WORDDATE, SALARY1+SALARY2 AS SALARY FROM USER
  WHERE DEGREE IN (3,4)
),
TEMP2 (DEGREE,WORDDATE,AVG_SALARY) AS
(
  SELECT DEGREE,WORDDATE,AVG(SALARY) AS AVG_SALARY
  FROM TEMP1
  GROUP BY DEGREE,WORDDATE
)
SELECT NAME FROM TEMP1, TEMP2 WHERE
TEMP1.DEGREE=TEMP2.DEGREE
AND TEMP1.WORDDATE=TEMP2.WORDDATE
AND SALARY<AVG_SALARY;
```


不用with的实现方法



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

With实现递归查询

```
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');
```

问题：查询‘DB2’下所有评论？

```
WITH TEMP(PARENTID,ID,NAME) AS
```

```
(
```

```
SELECT PARENTID,ID,NAME FROM BBS WHERE NAME='DB2'---语句 1
```

```
UNION ALL---语句 2
```

```
SELECT B.PARENTID,B.ID,B.NAME FROM BBS AS B, TEMP AS T WHERE B.PARENTI  
D=T.ID---语句 3
```

```
)
```

```
SELECT NAME FROM TEMP;---语句 4
```



- 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。

Values使用

- 定义临时集合，用明确的值定义临时集合

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

```
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
-----	----------

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




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



3、
`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';`

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



```
SELECT * FROM USER WHERE DEPARTMENT=1 AND BIRTHDAY='1949-10-1'
```

```
UNION
```

```
SELECT * FROM USER WHERE DEPARTMENT=2 AND BIRTHDAY='1978-12-18'
```

```
UNION
```

```
SELECT * FROM USER WHERE DEPARTMENT=3 AND BIRTHDAY='1997-7-1'
```



DB2 OLAP函数

在线分析处理的格式

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

ROW_NUMBER

RANK

DENSE_RANK

FIRST_VALUE

LAST_VALUE

LAG

LEAD

COUNT

MIN

MAX

AVG

SUM

- **Over**是表表达式，目的是定义一个结果集

- 根据工资排序，显示序号

```
SELECT
```

```
    ROW_NUMBER() OVER(ORDER BY SALARY) AS 序号,
```

```
    NAME AS 姓名,
```

```
    DEPT AS 部门,
```

```
    SALARY AS 工资
```

```
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);
```

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

- 追加对部门员工的平均工资
- 全体员工的平均工资

SELECT

```
ROW_NUMBER() OVER() AS 序号,
```

```
ROW_NUMBER() OVER(PARTITION BY DEPT ORDER BY SALARY) AS 部门序号,
```

```
NAME AS 姓名,
```

```
DEPT AS 部门,
```

```
SALARY AS 工资,
```

```
AVG(SALARY) OVER(PARTITION BY DEPT) AS 部门平均工资,
```

```
AVG(SALARY) OVER() AS 全员平均工资
```

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);
```

Partition by = group by



序号	部门序号	姓名	部门	工资	部门平均工资
全员平均工资					
1	1	张三	市场部	4000	4500
	4000				
2	2	李四	市场部	5000	4500
	4000				
3	3	王五	市场部	(null)	4500
	4000				
4	1	赵红	技术部	2000	3666
	4000				
5	2	王蓝	技术部	4000	3666
	4000				
6	3	李白	技术部	5000	3666
	4000				



```

SELECT
FROM_NUMBER() OVER(ORDER BY SALARY desc NULLS FIRST) AS RN,
RANK() OVER(ORDER BY SALARY desc NULLS FIRST) AS RK,
DENSE_RANK() OVER(ORDER BY SALARY desc NULLS FIRST) AS D_RK,
NAME AS 姓名,
DEPT AS 部门,
SALARY AS 工资
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);

```

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

分页查询

```
SELECT * FROM  
(  
    SELECT B.*, ROWNUMBER() OVER() AS RN FROM  
    (  
        SELECT * FROM EMPLOYEE ORDER BY EMPNO  
    ) AS B  
) AS A WHERE A.RN BETWEEN 20 AND 40 ;
```

DB2行转列

班级	科目	分数
----	----	----

1	语文	8800
---	----	------

1	数学	8420
---	----	------

1	英语	7812
---	----	------

.....

2	语文	8715
---	----	------

2	数学	8511
---	----	------

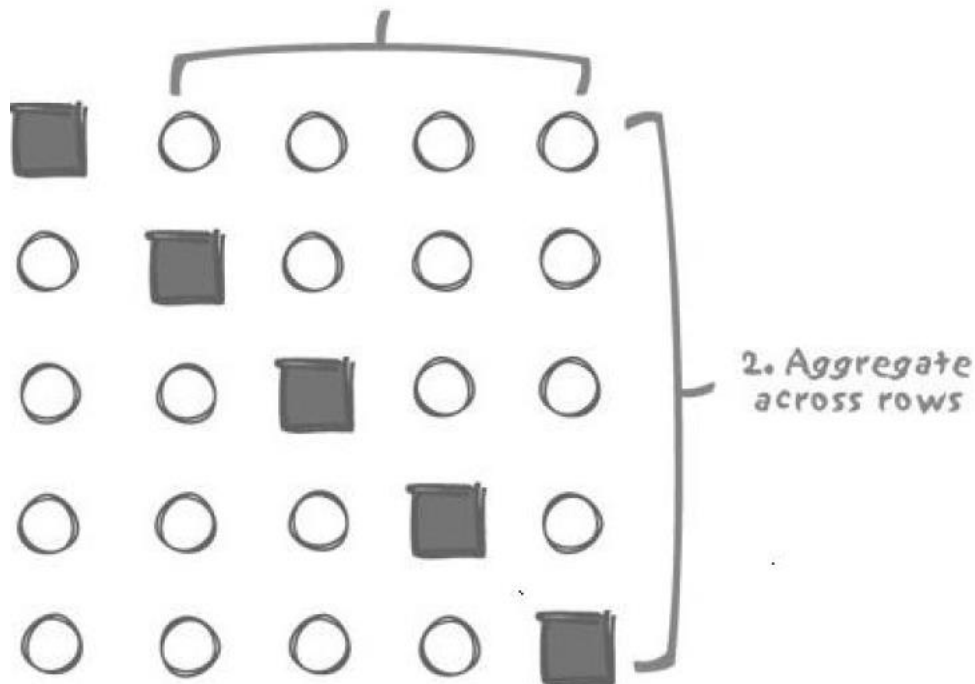
2	英语	8512
---	----	------


班级	语文	数学	英语
----	----	----	----

1	8800	8420	7812
---	------	------	------

2	8715	8511	8512
---	------	------	------

1. Complete each row with dummy values





```
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
```


类似行转列问题

ID	A	B
1	1	a
2	2	b
3	1	c
4	1	d
5	3	e
6	3	f

RA	RB
1	a, c, d
2	b
3	e, f

```
create function combin_str( v_id varchar(100) )  
returns varchar(1000)
```

```
language sql
```

```
begin atomic
```

```
    declare v_b varchar(1000) default ' ' ;
```

```
    for v_row as select b from t1 where id = v_id
```

```
    do
```

```
        set v_b = v_b || ',' || v_row.b ;
```

```
    end for;
```

```
    return v_b;
```

```
end
```

```
@
```

```
select distinct id, combin_str(id) from t1 order by id
```

某移动报表系统行转列

- 某移动的一张月报表执行一次需要7个小时以上，基本功能是实现行列数据转换，数据量为1300万行。简化如下：

Groupid, name

1001, 'aaa'

1001, 'bbb'

1001, 'cccc'

1002, 'bbccc'

1002, 'dddddd'

报表要求：

Groupid, totalnames

1001, 'aaa,bbb,cccc'

1002, 'bbccc,dddddd'

- 通过SP调整方案后，执行时间由7小时以上变为20分钟

Student exercise

