

Querying Data From A Table

```
SELECT c1, c2 * 2 AS "2倍c2", 5 FROM t
WHERE condition
ORDER BY cx DESC
LIMIT n;
```

- 获取表 `t` 中的列 `c1` , 2倍的列 `c2` 值, 常数 `5` 的数据 (如使用 `*` 则为获取所有列)。
- 可(选择性)使用关键字 `AS` 为列设定别名。使用中文别名需用双引号 `"` (不是单引号 `'`)。
- 可以在第一个列 `c1` 前使用 `DISTINCT` 可以删除重复行。(`DISTINCT` 关键字只能用在第一个列 `c1` 之前。不能在其他列之前)
- 通过 `WHERE` 子句来指定查询数据的条件。 `WHERE` 子句必须紧跟在 `FROM` 子句之后。首先通过 `WHERE` 子句查询出符合指定条件的记录, 然后再选取出 `SELECT` 语句指定的列。
- 是否为 `NULL` 要用 `IS (NOT) NULL` 运算符。

运算符	含义
<code>=</code>	和 ~ 相等
<code><></code>	和 ~ 不相等
<code>>=</code>	大于等于 ~
<code>></code>	大于 ~
<code><=</code>	小于等于 ~
<code><</code>	小于 ~

- `ORDER BY` 默认为 `ASC` 升序。不论何种情况, `ORDER BY` 子句都需要写在 `SELECT` 语句的末尾(`LIMIT` 前)。
- `LIMIT` 仅显示头 `n` 行数据。

```
SELECT c1, c2, 5, aggregate(c3)
FROM t
WHERE condition
GROUP BY c1, c2
HAVING condition
ORDER BY cx DESC;
```

- `aggregate()` 聚合函数:
 - `COUNT`: 计算表中的记录数(行数)。`COUNT(*)`会得到包含`NULL`的数据行数, 而`COUNT(<列名>)`会得到`NULL`之外的数据行数。
 - `SUM`: 计算表中数值列中数据的总和值(只能数值型列)
 - `AVG`: 计算表中数值列中数据的平均值(只能数值型列)
 - `MAX`: 求出表中任意列中数据的最大值
 - `MIN`: 求出表中任意列中数据的最小值
- 不使用 `GROUP BY` 子句时, 是将表中的所有数据作为一组来对待的。而使用 `GROUP BY` 子句时, 会将表中的数据分为多个组进行处理。
- 使用 `GROUP BY`时, `SELECT` 子句中只能存在以下三种元素。
 - 常数

- 聚合函数
- **GROUP BY** 子句中指定的列名（也就是聚合键）
- 使用 聚合函数 和 **GROUP BY** 常见错误:
 - 把聚合键之外的列名，写在 **SELECT** 子句之中。即除了 **GROUP BY** 中的 **c1, c2** 外，不能有其他列。
 - 在 **GROUP BY** 子句中不能使用 **SELECT** 子句中定义的别名。
 - **GROUP BY** 子句结果的显示是无序的。
 - 只有 **SELECT** 子句和 **HAVING** 子句（以及 **ORDER BY** 子句）中能够使用聚合函数。
- **HAVE** 子句中能够使用的3种要素如下所示(和 **WHERE** 不同，不能随意用列运算符)。
 - 常数
 - 聚合函数
 - **GROUP BY**子句中指定的列名（即聚合键）

WHERE 子句 = 指定行所对应的条件

HAVE 子句 = 指定组所对应的条件

子句书写顺序

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

程序执行顺序

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

子查询

子查询就是将用来定义视图的 **SELECT** 语句直接用于 **FROM, WHERE** 子句当中。

SELECT c1, c2, 5, **aggregate(c3)**

FROM (**SELECT** cx, cy
 FROM t1);

SELECT c1, c2, 5

FROM t

WHERE condition1 > (**SELECT** **aggregate(c3)**
 FROM t1);

SELECT c1, c2,

 (**SELECT** **aggregate(c3)**

FROM t1) **AS** 别名1

FROM t

SELECT c1, **aggregate(c3)**

FROM t

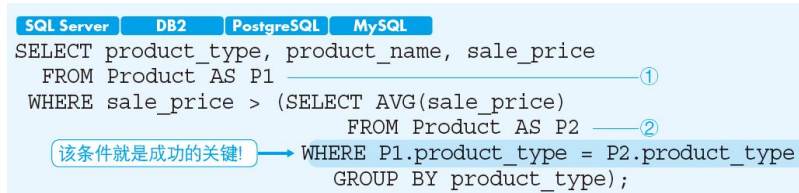
GROUP BY c1

HAVING **aggregate(c3)** > (**SELECT** **aggregate(c3)**
 FROM t1)

使用子查询的 SQL 会从子查询开始执行。

标量(单一值)子查询的书写位置并不仅仅局限于 **WHERE** 子句中，通常任何可以使用单一值的位置都可以使用。也就是说，能够使用常数或者列名的地方，无论是 **SELECT** , **GROUP BY**, **HAVING** 子句，还是 **ORDER BY** 子句，几乎所有的地方都可以使用。(注意：使用标量子查询时，该子查询 绝对不能返回多行结果。)

[关联子查询](<https://zhuanlan.zhihu.com/p/41844742>):



```
SQL Server DB2 PostgreSQL MySQL
SELECT product_type, product_name, sale_price
  FROM Product AS P1
 WHERE sale_price > (SELECT AVG(sale_price)
                     FROM Product AS P2
                     WHERE P1.product_type = P2.product_type
                     GROUP BY product_type);
```

关联子查询和正常的SELECT语句完全不同。

1. 先执行主查询

```
SELECT product_type , product_name, sale_price
FROM Product AS P1
```

1. 从主查询的 product_type 先取第一个值="衣服"，通过 WHERE P1.product_type = P2.product_type 传入子查询，子查询变成：

```
SELECT AVG(sale_price)
FROM Product AS P2
WHERE P2.product_type = "衣服"
GROUP BY product_type);
```

1. 从子查询得到的结果 AVG(sale_price)=2500，返回主查询：

```
SELECT product_type , product_name, sale_price
FROM Product AS P1
WHERE sale_price > 2500 AND product_type = '衣服'
```

1. 然后，product_type 取第二个值，得到整个语句的第二结果，依次类推，把 product_type 全取值一遍，就得到了整个语句的结果集。

注意：关联/结合条件一定要写在子查询中。

使用谓词

LIKE - 模糊查询。同正则表达式一起使用，用正则符号来替换字符串。

- **"%"**: 表示"0 字符以上的任意字符串"的特殊符号。
 - 找出在列 c1 中以 ddd 结尾的所有行：

```
SELECT *
FROM t
WHERE c1 LIKE '%ddd';
```
- **"_"**(下划线): 表示"任意1 个字符"。
 - 找出在列 c1 中以 abc+任意2个字符的所有行：

```
SELECT *
FROM t
WHERE c1 LIKE 'abc__';
```

BETWEEN - 范围查询(包含两个临界值)。如选取销售单价为100 ~ 1000元的商品

- **SELECT** product_name, sale_price
FROM Product
WHERE sale_price **BETWEEN** 100 **AND** 1000;

IS NULL / IS NOT NULL — 判断是否为 NULL (判断 NULL 不能用 =)。如选取出销售单价为 NULL 的商品

- **SELECT** product_name, sale_price
FROM Product
WHERE sale_price **IS NULL**;

IN / NOT IN - 通过 IN 或者 NOT IN来指定 或者 排除 多个数值(v1, v2, ...)进行查询(使用 IN 和 NOT IN 是无法选取出 NULL 数据的。)

- **SELECT** c1, c2
FROM t
WHERE c1 **IN** (v1, v2, v3,...);

IN / NOT IN - 子查询生成的表，能作为 IN 或者 NOT IN 的参数

- **SELECT** c1, c2
FROM t
WHERE c1 **IN** (**SELECT**...);

集合运算

- **UNION** 将输出表的内容合在一起
- **INTERSECT** 选取表中公共部分
- **EXCEPT** 从第一张生成表内，剔除和第二张生成表公共部分

语法：将以上关键词放在两个 **SELECT** 子句(生成表)之间。

- **SELECT** c1, c2
FROM t1
UNION / INTERSECT / EXCEPT
SELECT c1, c2
FROM t2

注意事项：

- 集合运算符会除去重复的记录。(使用 **UNION ALL** 可包含重复行)
- 作为运算对象的记录的列数必须相同(不能一部分记录包含2列，另一部分记录包含3列)
- 作为运算对象的记录中列的类型必须一致
- 可以使用任何 **SELECT** 语句，但 **ORDER BY** 子句只能在最后使用一次

表联结 (**JOIN**)

内联结 (**INNER JOIN**)

通过不同表格中的相同列，根据这些相同列中相同的 cell 进行联结，汇集成一张表格。

- 进行联结时需要在 **FROM** 子句中使用多张表。
- 进行内联结时必须使用 **ON** 子句，并且要书写在 **FROM** 和 **WHERE** 之间。
- 使用联结时 **SELECT** 子句中的列需要按照 "<表的别名>.<列名>" 的格式进行书写。
- 使用联结运算将满足相同规则的表联结起来时，**WHERE**, **GROUP BY**, **HAVING**, **ORDER BY** 等工具都可以正常使用。

如：从 Product 表中取出商品名称(product_name)和销售单价(sale_price)，并与 ShopProduct 表中的内容进行结合。且只想知道'000A'店的信息

- **SELECT** SP.shop_id, SP.shop_name, SP.product_id, P.product_name, P.sale_price
FROM ShopProduct **AS** SP **INNER JOIN** Product **AS** P
ON SP.product_id = P.product_id
WHERE SP.shop_id = '000A';

外联结 (OUTER JOIN / LEFT JOIN / RIGHT JOIN)

以主表为主体，通过不同表格中和主表指定的相同列，同主表指定列中所有 cell 进行联结，汇集成一张表格。

- 选取出主表中全部的信息
- 外联结中使用 **LEFT**, **RIGHT** 来指定主表。使用 **LEFT** 时 **FROM** 子句中写在左侧的表是主表，使用 **RIGHT** 时右侧的表是主表。

如：从 Product 表中取出商品名称(product_name)和销售单价(sale_price)，并与 ShopProduct 表中的内容进行外结合。且只想知道'000A'店的信息

- **SELECT** SP.shop_id, SP.shop_name, SP.product_id, P.product_name, P.sale_price
FROM ShopProduct **AS** SP **LEFT JOIN** Product **AS** P
ON SP.product_id = P.product_id
WHERE SP.shop_id = '000A';

对多表进行内联结

```
SELECT SP.shop_id, SP.shop_name, SP.product_id, P.product_name, P.sale_price, IP.inventory_id
FROM ShopProduct AS SP INNER JOIN Product AS P
ON SP.product_id = P.product_id
    INNER JOIN InventoryProduct AS IP
    ON SP.product_id = IP.product_id
WHERE IP.inventory_id = 'P001';
```

Managing Data

创建数据库:

```
CREATE DATABASE D_name;
```

创建表:

```
CREATE TABLE t_name (  
c1 INT PRIMARY KEY,  
c2 VARCHAR NOT NULL,  
c3 FLOAT DEFAULT 0  
);
```

或者:

```
CREATE TABLE t_name (  
c1 INT NOT NULL,  
c2 VARCHAR NOT NULL,  
c3 FLOAT DEFAULT 0,  
PRIMARY KEY (c1)  
);
```

删除整个表:

```
DROP TABLE t_name;
```

删除表内数据, 保留 **Header**:

```
TRUNCATE TABLE t_name;
```

或者删除指定数据行:

```
DELETE FROM t_name  
WHERE condition
```

- **DELETE** 可以通过 **WHERE** 子句指定对象条件来删除部分数据。
- **TRUNCATE** 只能删除表中的全部数据, 而不能通过 **WHERE** 子句指定条件来删除部分数据。

更新表定义: (如果是表中数值, 则用 **UPDATE**)

添加列:

```
ALTER TABLE t_name ADD COLUMN c1;
```

删除列:

```
ALTER TABLE t_name DROP COLUMN c1;
```

更改列名:

```
ALTER TABLE t_name RENAME c1 TO c2;
```

变更表名:

- PostgreSQL/Oracle: **ALTER TABLE t1 RENAME TO t2;**
- MySQL: **RENAME TABLE t1 TO t2;**
- SQL_Server: **SP_RENAME 't1', 'Product';**

Modifying Data

插入数据:

```
INSERT INTO t_name (c1, c2, ...)
VALUES (v1, v2, ...),
        (x1, x2, ...),
        ...
        (z1, z2, ...);
```

在表 t_name 中，对应列 c1, c2, ... 依次插入多行数据。插入一行只需保留(v1, v2, ...)，删除其他 VALUES。

- 在 **VALUES** 子句中使用 **DEFAULT** 指定该对应列使用默认值。
- 省略 **INSERT** 语句中的列名，就会自动设定为该列的默认值(没有默认值时会设定为 **NULL**)。

从其他表中复制数据:

```
INSERT INTO t1 (c1, c2, ...)
SELECT cx, aggregate(cy), ...
FROM t2
```

INSERT INTO语句的 **SELECT** 语句中，可以使用 **WHERE** 子句或者 **GROUP BY** 子句等任何SQL语法（但使用 **ORDER BY** 子句并不会产生任何效果）。

*改变表中已有数值: (可使用指定条件 **WHERE**，更新部分数据行)*

```
UPDATE t1
SET cx = 数值 或者 表达式 ,
      cy = 数值 或者 表达式
WHERE condition ;
```

Managing VIEWS

视图保存的是 **SELECT** 语句，但是视图不保存数据。从视图中读取数据时，视图会在内部执行该 **SELECT** 语句并创建出一张临时表。应该将经常使用的 **SELECT** 语句做成视图。

创建视图:

```
CREATE VIEW v (c1,c2)  
AS  
SELECT 语句... ;
```

定义视图时可以使用任何 **SELECT** 语句，既可以使用 **WHERE**、**GROUP BY**、**HAVING** (但不能使用 **ORDER BY**)，也可以通过 **SELECT *** 来指定全部列。

应该避免在视图的基础上创建视图。

使用视图:

视图和表一样，可以书写在 **SELECT** 语句的 **FROM** 子句之中。

```
SELECT c1, c2, ...  
FROM view_name, ... ;
```

通过汇总(如 **GROUP BY**)得到的视图无法进行更新(使用 **INSERT**，**DELETE**，**UPDATE**)。

删除视图:

```
DROP VIEW v1, v2;  
来删除视图。
```


函数

算术函数（用来进行数值计算的函数）

- 绝对值: [ABS\(列名\)](#)
- 余数: [MOD\(被除数,除数\)](#) -- 被除数和除数都为列名。
 - [SQL Server] - 使用 [%"](#) 来计算余数)
- 四舍五入: [ROUND\(对象数值,保留小数的位数\)](#)

字符串函数（用来进行字符串操作的函数）

- 拼接: [str1 || str2](#) -- 实现 `abc+de = abcde`.
 - [SQL Server] - 对字符串使用 ["+"](#)
 - [MySQL] - 使用函数 [CONCAT\(str1, str2, str3...\)](#)
- 字符串长度: [LENGTH\(str\)](#)
 - [SQL Server] - [LEN\(str\)](#)
- 大 / 小写转换: [UPPER\(str\)](#) / [LOWER\(str\)](#)
- 字符串的替换: [REPLACE\(str1, str2, str3\)](#) -- 将 `str1` 中含有的 `str2` 字符, 转换成 `str3`。
- 字符串的截取:
 - [PostgreSQL / MySQL] - [SUBSTRING\(对象字符串 FROM 截取的起始位置 FOR 截取的字符数\)](#)
 - [SQL Server] - [SUBSTRING\(对象字符串, 截取的起始位置, 截取的字符数\)](#)
 - [Oracle 和 DB2] - [SUBSTR\(对象字符串, 截取的起始位置, 截取的字符数\)](#)

日期函数（用来进行日期操作的函数）

- 当前日期:
 - [MySQL, PostgreSQL] - [CURRENT_DATE](#)
 - [SQL Server] - [CURRENT_TIMESTAMP](#)
 - 使用 [CAST](#) 函数将 [CURRENT_TIMESTAMP](#) 转换为时间类型
 - [SELECT CAST\(CURRENT_TIMESTAMP AS TIME\) AS CUR_TIME;](#)
 - [Oracle] (需要在 [FROM](#) 子句中指定临时表([DUAL](#))) - [SELECT CURRENT_DATE FROM DUAL;](#)
 - [DB2] - [SELECT CURRENT DATE FROM SYSIBM.SYSDUMMY1;](#)
- 当前时间:
 - [MySQL, PostgreSQL] - [CURRENT_TIME](#)
 - [SQL Server] - [CURRENT_TIMESTAMP](#)
 - 使用 [CAST](#) 函数将 [CURRENT_TIMESTAMP](#) 转换为日期类型
 - [SELECT CAST\(CURRENT_TIMESTAMP AS DATE\) AS CUR_DATE;](#)
 - [Oracle] (需要在 [FROM](#) 子句中指定临时表([DUAL](#))) - [SELECT CURRENT_TIMESTAMP FROM DUAL;](#)

- [DB2] - [SELECT CURRENT TIME FROM SYSIBM.SYSDUMMY1;](#)
- 当前日期和时间: [CURRENT_TIMESTAMP](#) (函数具有 CURRENT_DATE + CURRENT_TIME 的功能。使用该函数可以同时得到当前的日期和时间)
 - [SQL Server, PostgreSQL, MySQL] - [SELECT CURRENT_TIMESTAMP;](#)
 - [Oracle] - [SELECT CURRENT_TIMESTAMP FROM DUAL;](#)
 - [DB2] - [SELECT CURRENT TIMESTAMP FROM SYSIBM.SYSDUMMY1;](#)
- 截取日期元素:
 - [PostgreSQL, MySQL] - [EXTRACT\(日期元素 FROM 日期\)](#)
 - SELECT CURRENT_TIMESTAMP,
EXTRACT(YEAR FROM CURRENT_TIMESTAMP) AS year,
EXTRACT(MONTH FROM CURRENT_TIMESTAMP) AS month,
EXTRACT(DAY FROM CURRENT_TIMESTAMP) AS day,
EXTRACT(HOUR FROM CURRENT_TIMESTAMP) AS hour,
EXTRACT(MINUTE FROM CURRENT_TIMESTAMP) AS minute,
EXTRACT(SECOND FROM CURRENT_TIMESTAMP) AS second;
 - [SQL Server] - [DATEPART\(日期元素, 日期\)](#)
 - SELECT CURRENT_TIMESTAMP,
DATEPART(YEAR , CURRENT_TIMESTAMP) AS year,
DATEPART(MONTH , CURRENT_TIMESTAMP) AS month,
DATEPART(DAY , CURRENT_TIMESTAMP) AS day,
DATEPART(HOUR , CURRENT_TIMESTAMP) AS hour,
DATEPART(MINUTE , CURRENT_TIMESTAMP) AS minute,
DATEPART(SECOND , CURRENT_TIMESTAMP) AS second;
 - [Oracle, DB2] - [EXTRACT\(日期元素 FROM 日期\)](#)
 - SELECT CURRENT_TIMESTAMP,
EXTRACT(YEAR FROM CURRENT_TIMESTAMP) AS year,
EXTRACT(MONTH FROM CURRENT_TIMESTAMP) AS month,
EXTRACT(DAY FROM CURRENT_TIMESTAMP) AS day,
EXTRACT(HOUR FROM CURRENT_TIMESTAMP) AS hour,
EXTRACT(MINUTE FROM CURRENT_TIMESTAMP) AS minute,
EXTRACT(SECOND FROM CURRENT_TIMESTAMP) AS second
FROM DUAL; / FROM SYSIBM.SYSDUMMY1;

转换函数（用来转换数据类型和值的函数）

- 类型转换: [CAST\(转换前的值 AS 想要转换的数据类型\)](#)
 - [SQL Server, MySQL, PostgreSQL] - 字符串转日期类型
 - SELECT CAST('2009-12-14' AS DATE) AS date_col;
 - [Oracle, DB2] - 字符串转日期类型
 - SELECT CAST('2009-12-14' AS DATE) AS date_col FROM DUAL; / FROM SYSIBM.SYSDUMMY1;

聚合函数（用来进行数据聚合的函数）

- 聚合函数基本上只包含 [COUNT](#), [SUM](#), [AVG](#), [MAX](#), [MIN](#) 这5种。

SQL 高级处理

窗口函数 (OLAP - OnLine Analytical Processing)

OLAP 函数也称为窗口函数。为了实现对数据库数据进行实时分析处理，而添加的标准 SQL 功能。例如，市场分析、创建财务报表、创建计划等日常性商务工作。

窗口函数就是将表以窗口为单位进行分割，并在其中进行排序的函数。

窗口函数 OVER (PARTITION BY c1 ORDER BY c2)

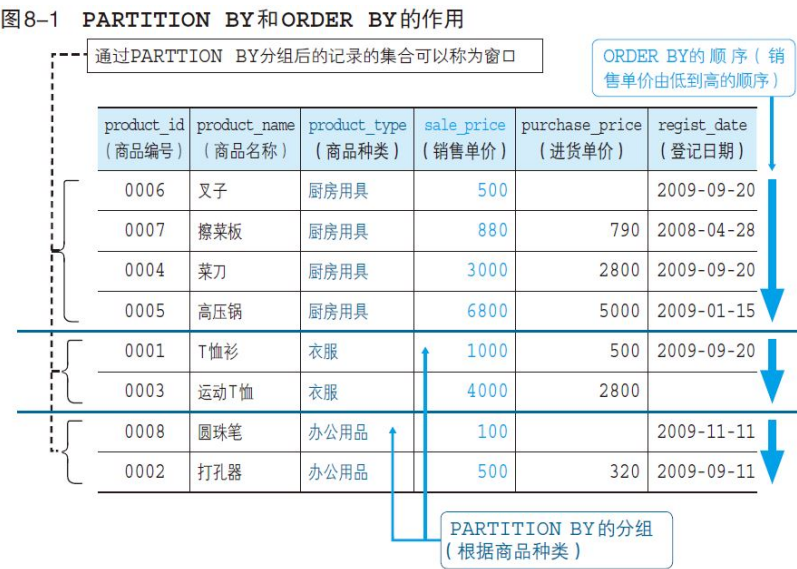
PARTITION BY 指定按照哪个列(对象)来进行排序, 在横向上对表进行分组。但不具备 GROUP BY 的汇总功能。ORDER BY 能够指定按照哪一行, 何种顺序进行排序, 决定了纵向排序的规则。

- 能够作为窗口函数的聚合函数（SUM、AVG、COUNT、MAX、MIN）
- RANK(), DENSE_RANK(), ROW_NUMBER() 等专用窗口函数。由于专用窗口函数无需参数，因此通常括号中都是空的。
- 原则上窗口函数只能在 SELECT 子句中使用。
- 使用窗口函数时，在 OVER 子句中必须使用 ORDER BY。OVER 子句中的 ORDER BY 只是用来决定窗口函数按照什么样的顺序进行计算的，对结果的排列顺序并没有影响。
- 让记录切实按照窗口函数产生的列排序，还是需要在 SELECT 语句的最后，使用 ORDER BY 子句进行指定。

使用专用窗口函数

如根据不同的商品种类，按照销售单价从低到高的顺序创建排序表：

```
SELECT product_name, product_type, sale_price,  
RANK () OVER (PARTITION BY product_type  
ORDER BY sale_price) AS ranking  
FROM Product  
ORDER BY sale_price;
```



无需指定 **PARTITION BY**。整个表作为一个大的窗口来使用。(与直接使用 **ORDER BY** 类似)
所以当希望先将表中的数据分为多个部分（窗口），再使用窗口函数时，可以使用 **PARTITION BY** 选项。

- **RANK()**： 计算排序时，如果存在相同位次的记录，则会跳过之后的位次。例）有3条记录排在第1位时： 1位, 1位, 1位, 4位...
- **DENSE_RANK()**：即使存在相同位次的记录，也不会跳过之后的位次。例）有3条记录排在第1位时： 1位, 1位, 1位, 2位...
- **ROW_NUMBER()**：赋予唯一的连续位次。例）有3条记录排在第1位时： 1位, 2位, 3位, 4位...

代码清单 8-3 比较 **RANK**、**DENSE_RANK**、**ROW_NUMBER** 的结果

```
Oracle SQL Server DB2 PostgreSQL
SELECT product_name, product_type, sale_price,
       RANK () OVER (ORDER BY sale_price) AS ranking,
       DENSE_RANK () OVER (ORDER BY sale_price) AS dense_ranking,
       ROW_NUMBER () OVER (ORDER BY sale_price) AS row_num
FROM Product;
```

执行结果

product_name	product_type	sale_price	RANK	DENSE_RANK	ROW_NUMBER
			ranking	dense_ranking	row_num
圆珠笔	办公用品	100	1	1	1
叉子	厨房用具	500	2	2	2
打孔器	办公用品	500	2	2	3
擦菜板	厨房用具	880	4	3	4
T恤衫	衣服	1000	5	4	5
菜刀	厨房用具	3000	6	5	6
运动T恤	衣服	4000	7	6	7
高压锅	厨房用具	6800	8	7	8

使用聚合函数作为窗口函数

以“自身记录（当前记录）”作为基准进行统计，是将聚合函数当作窗口函数使用时的最大特征。

如使用 **SUM** 函数: 该合计值的逻辑是逐行添加计算对象。在按照时间序列的顺序，计算各个时间的销售额总额等的时候，通常都会使用这种称为累计的统计方法。

代码清单 8-4 将 **SUM** 函数作为窗口函数使用

```
Oracle SQL Server DB2 PostgreSQL
SELECT product_id, product_name, sale_price,
       SUM (sale_price) OVER (ORDER BY product_id) AS current_sum
FROM Product;
```

执行结果

product_id	product_name	sale_price	current_sum
0001	T恤衫	1000	1000
0002	打孔器	500	1500
0003	运动T恤	4000	5500
0004	菜刀	3000	8500
0005	高压锅	6800	15300
0006	叉子	500	15800
0007	擦菜板	880	16680
0008	圆珠笔	100	16780

代码清单 8-5 将 **AVG** 函数作为窗口函数使用

```
Oracle SQL Server DB2 PostgreSQL
SELECT product_id, product_name, sale_price,
       AVG (sale_price) OVER (ORDER BY product_id) AS current_avg
FROM Product;
```

product_id	product_name	sale_price	current_avg
0001	T恤衫	1000	1000.00000000000000000000 ←(1000)/1
0002	打孔器	500	750.00000000000000000000 ←(1000+500)/2
0003	运动T恤	4000	1833.33333333333333333333 ←(1000+500+4000)/3
0004	菜刀	3000	2125.00000000000000000000 ←(1000+500+4000+3000)/4
0005	高压锅	6800	3060.00000000000000000000 ←(1000+500+4000+3000+6800)/5
0006	叉子	500	2633.33333333333333333333 •
0007	擦菜板	880	2382.8571428571428571 •
0008	圆珠笔	100	2097.50000000000000000000 •

指定框架，需要在 **ORDER BY** 子句之后使用指定范围的关键字 **ROWS**（“行”），**PRECEDING**（“之前”），**FOLLOWING**（“之后”）。

```
SELECT product_id, product_name, sale_price,
       AVG (sale_price) OVER (ORDER BY product_id
                               ROWS 2 PRECEDING) AS moving_avg
FROM Product;
```

ROWS 2 PRECEDING

product_id (商品编号)	product_name (商品名称)	sale_price (销售单价)
0001	T恤衫	1000
0002	打孔器	500
0003	运动T恤	4000
0004	菜刀	3000
0005	高压锅	6800
0006	叉子	500
0007	擦菜板	880
0008	圆珠笔	100

如需将当前记录的前后行作为汇总对象，如下同时使用**PRECEDING**（“之前1行”），**FOLLOWING**（“之后1行”）和本身进行计算。

```
SELECT product_id, product_name, sale_price,  
       AVG (sale_price) OVER (ORDER BY product_id  
                               ROWS BETWEEN 1 PRECEDING AND  
                                       1 FOLLOWING) AS moving_avg  
FROM Product;
```

GROUPING

只使用 **GROUP BY** 子句和聚合函数是无法同时得出合计行和小计的。
使用 **GROUPING** 运算符(包含**ROLLUP**, **CUBE**, **GROUPING SETS**)可以同时得到。如下图：

合计	16780	←存在合计行
厨房用具	11180	
衣服	5000	
办公用品	600	

ROLLUP 同时得出合计和小计

OracleSQL ServerDB2PostgreSQL

```
SELECT product_type, regist_date, SUM(sale_price) AS sum_price
FROM Product
GROUP BY ROLLUP(product_type, regist_date); ①
```

在 MySQL 中执行以上代码，请将①中的 **GROUP BY** 子句改写为 “**GROUP BY product_type, regist_date WITH ROLLUP;**”

product_type	regist_date	sum_price	
		16780	←合计
厨房用具		11180	←小计（厨房用具）
厨房用具	2008-04-28	880	
厨房用具	2009-01-15	6800	
厨房用具	2009-09-20	3500	
办公用品		600	←小计（办公用品）
办公用品	2009-09-11	500	
办公用品	2009-11-11	100	
衣服		5000	←小计（衣服）
衣服	2009-09-20	1000	
衣服		4000	

GROUPING 函数判断 NULL

下面代码中，当 **GROUPING** 判断为 **NULL**，则插入“合计”或者“小计”等字符串，其他情况返回通常的列的值。

代码清单 8-16 在超级分组记录的键值中插入恰当的字符串

OracleSQL ServerDB2PostgreSQL

```
SELECT CASE WHEN GROUPING(product_type) = 1
          THEN '商品种类 合计'
          ELSE product_type END AS product_type,
       CASE WHEN GROUPING(regist_date) = 1
          THEN '登记日期 合计'
          ELSE CAST(regist_date AS VARCHAR(16)) END AS regist_date,
       SUM(sale_price) AS sum_price
FROM Product
GROUP BY ROLLUP(product_type, regist_date);
```

执行结果 (在DB2中执行)

product_type	regist_date	sum_price
-----	-----	-----
商品种类 合计	登记日期 合计	16780
厨房用具	登记日期 合计	11180
厨房用具	2008-04-28	880
厨房用具	2009-01-15	6800
厨房用具	2009-09-20	3500
办公用品	登记日期 合计	600
办公用品	2009-09-11	500
办公用品	2009-11-11	100
衣服	登记日期 合计	5000
衣服	2009-09-20	1000
衣服		4000

将超级分组记录中的NULL
替换为“登记日期 合计”
原始数据中的NULL保持
不变

使用 **CAST** 是为了满足 **CASE** 表达式所有分支的返回值必须一致的条件。如果不这样的话，那么各个分支会分别返回日期类型和字符串类型的值，执行时就会发生语法错误。

CUBE

所谓 **CUBE**，就是将 **GROUP BY** 子句中聚合键的“所有可能的组合”的汇总结果集中到一个结果中。如下图，多出来的记录就是只把 **regist_date** 作为聚合键所得到的汇总结果。

代码清单 8-17 使用 CUBE 取得全部组合的结果

OracleSQL ServerDB2PostgreSQL

```
SELECT CASE WHEN GROUPING(product_type) = 1
          THEN '商品种类 合计'
          ELSE product_type END AS product_type,
       CASE WHEN GROUPING(regist_date) = 1
          THEN '登记日期 合计'
          ELSE CAST(regist_date AS VARCHAR(16)) END AS regist_date,
       SUM(sale_price) AS sum_price
FROM Product
GROUP BY CUBE(product_type, regist_date);
```

执行结果 (在DB2中执行)

product_type	regist_date	sum_price
-----	-----	-----
商品种类 合计	登记日期 合计	16780
商品种类 合计	2008-04-28	880
商品种类 合计	2009-01-15	6800
商品种类 合计	2009-09-11	500
商品种类 合计	2009-09-20	4500
商品种类 合计	2009-11-11	100
商品种类 合计		4000
厨房用具	登记日期 合计	11180
厨房用具	2008-04-28	880
厨房用具	2009-01-15	6800
厨房用具	2009-09-20	3500
办公用品	登记日期 合计	600
办公用品	2009-09-11	500
办公用品	2009-11-11	100
衣服	登记日期 合计	5000
衣服	2009-09-20	1000
衣服		4000

←追加
←追加
←追加
←追加
←追加
←追加