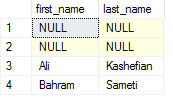


* SELECT

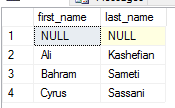
1. TOP n ; TOP(N) ; TOP N Percent; TOP(N) Percent
2. DISTINCT TOP n; DISTINCT TOP(N)

SELECT DISTINCT TOP(4) PERCENT \* FROM acct\_users

ORDER BY first\_name

SELECT TOP(4) first\_name, last\_name FROM acct\_users

ORDER BY first\_name

SELECT DISTINCT TOP(4) first\_name, last\_name FROM acct\_users

ORDER BY first\_name

1. WHERE … ORDER BY …

Where clause is not allowed to use alias column name;

SELECT left(last\_name, 2) AS cutname, last\_name FROM acct\_users

WHERE cutname > 'a' error

Order by is allowed to use alias column name

SELECT left(last\_name, 2) AS cutname, last\_name FROM acct\_users

ORDER BY cutname ok

1. GROUP BY … HAVING …

HAVING must come by GROUP BY

GROUP BY and HAVING are not allowed alias name

SELECT left(last\_name, 2) AS cutname, COUNT(0) AS CNT FROM acct\_users

WHERE last\_name > 'b' AND first\_name < 's' - filter original recordset: table scan

GROUP BY left(last\_name, 2)

HAVING left(last\_name, 2) > 'a' - only filter recordset after group

ORDER BY cutname DESC - ORDER BY apply to final recordset

1. OFFSET m [ROWS|ROW] FETCH [FIRST|NEXT] n [ROWS|ROW] ONLY

GOOD PAGING SOLUTION

Notes: 1) must have ORDER BY Clause

ROWS | ROW interchange

FIRST | NEXT interchange

Keyword: OFFSET … FETCH … ONLY is required

SELECT left(last\_name, 2) AS cutname, COUNT(0) AS CNT FROM acct\_users

WHERE last\_name > 'b' AND first\_name < 's'

GROUP BY left(last\_name, 2)

HAVING left(last\_name, 2) > 'a'

ORDER BY cutname DESC

OFFSET 2 ROW FETCH FIRST 3 ROW ONLY

* OVER
  1. ROW\_NUMBER()
  2. RANK()
  3. DENSE\_RANK()
  4. LEAD() – 取排序后面的
  5. LAG() – 取排序前面的

lag 和lead 有三个参数：

第一个参数是列名；

第二个参数是偏移的offset；

第三个参数是 超出记录窗口时的默认值；

* 1. 各种聚集函数：SUM(), COUNT(), AVG(), MIN(), MAX() ….

OVER(PARTITION BY c1[,c2..] ORDER BY d1[,d2..] [ASC|DESC])

PARTITION BY - 分组的

ORDER BY - 排序的依据， 指定的字段，可以和PARTITION BY 完全不一样

SELECT

LEFT(last\_name, 1) as LN,

ROW\_NUMBER() OVER(PARTITION BY LEFT(last\_name, 1) ORDER BY id DESC) AS rowid,

id,

last\_name

FROM acct\_users

SELECT

LEFT(last\_name, 1) as LN,

ROW\_NUMBER() OVER(PARTITION BY LEFT(last\_name, 1) ORDER BY id DESC) AS rowid,

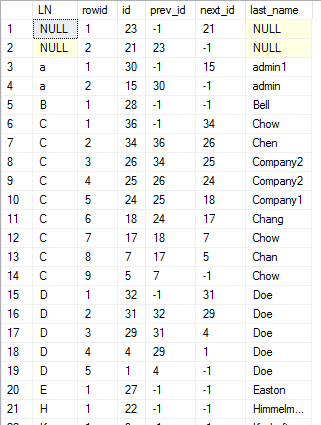
id,

LAG(id, 1, -1) OVER(PARTITION BY LEFT(last\_name, 1) ORDER BY id DESC) AS prev\_id,

LEAD(id, 1, -1) OVER(PARTITION BY LEFT(last\_name, 1) ORDER BY id DESC) AS next\_id,

last\_name

FROM acct\_users



 聚集函数：

SELECT id, name, month, salary,

SUM(salary) OVER(ORDER BY name, month) as Income

FROM employee

ORDER BY name, month

Keyword can be used: CURRENT ROW; UNBOUNDED PRECEDING; UNBOUNDED FOLLOWING

* ROWS UNBOUNDED PRECEDING vs RANGE UNBOUNDED PRECEDING

SELECT

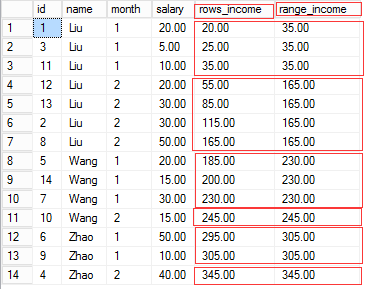
id, name, month, salary,

SUM(salary) OVER(ORDER BY name, month ROWS UNBOUNDED PRECEDING) as rows\_income,

SUM(salary) OVER(ORDER BY name, month RANGE UNBOUNDED PRECEDING ) as range\_income

FROM employee

ORDER BY name, month



SELECT

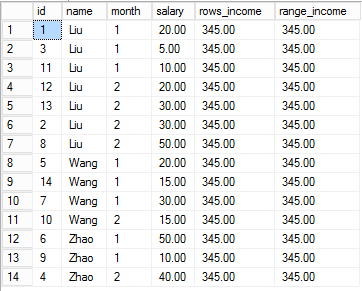
id, name, month, salary,

SUM(salary) OVER(ORDER BY name,month ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as rows\_income,

SUM(salary) OVER(ORDER BY name,month RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING ) as range\_income

FROM employee

ORDER BY name, month



由于没有指定 分区 partition by

ROWS 是指不分上下界

RANGE 也是不分上下界

SELECT

id, name, month, salary,

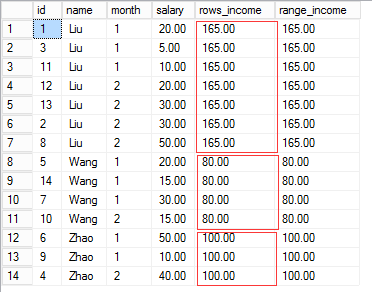
SUM(salary) OVER(PARTITION BY name ORDER BY name,month ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as rows\_income,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING ) as range\_income

FROM employee

ORDER BY name, month

我们指定了分区 PARTITION BY name



在分区内 ROWS RANGE 不分上下界

SELECT

id, name, month, salary,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month ROWS BETWEEN CURRENT ROW AND CURRENT ROW) as rows\_income,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month RANGE BETWEEN CURRENT ROW AND CURRENT ROW ) as range\_income

FROM employee

ORDER BY name, month



完整的例子：

SELECT

id, name, month, salary,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) as rows\_income1,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING ) as rows\_income2,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING ) as rows\_income3,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) as range\_income1,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING ) as range\_income2,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING ) as range\_income3

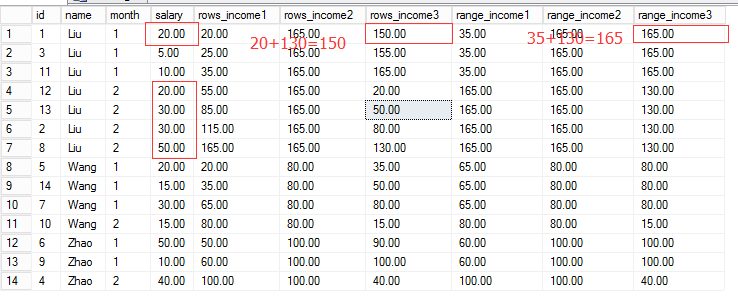
FROM employee

ORDER BY name, month

ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

看看一下的算法就知道如何工作的



SELECT

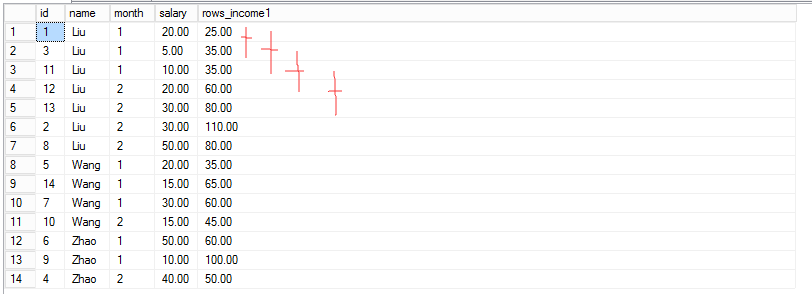
id, name, month, salary,

SUM(salary) OVER(PARTITION BY name ORDER BY name, month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING ) as rows\_income1

FROM employee

ORDER BY name, month

在分区内： 当前+前一个+后一个



ROWS BETWEEN 1 PRECEDING AND CURRENT ROW 分区内 前一个 + 当前

ROWS BETWEEN 1 PRECEDING AND UNBOUNDED FOLLOWING 分区内不知如何工作，奇怪？

ROWS BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING 分区内： 前面所有 + 当前 + 下一个

SELECT

id, name, month, salary,

SUM(salary) OVER(PARTITION BY name,MONTH ORDER BY name, month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as rows\_income1,

SUM(salary) OVER(PARTITION BY name,MONTH ORDER BY name, month ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) as rows\_income2

FROM employee

ORDER BY name, MONTH

结果一样？？



**ROWS vs RANGE**

**RANGE 是根据 ORDER BY 的排序条件， 也就是相同 ORDER BY 的行作为一个范围**

**ROWS 则是根据指定的行**

SELECT

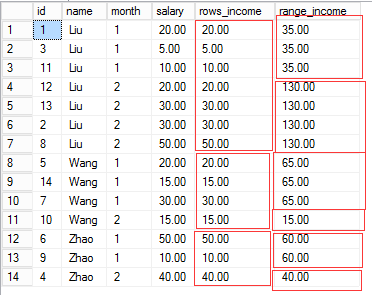
id, name, month, salary,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month ROWS BETWEEN CURRENT ROW AND CURRENT ROW) as rows\_income,

SUM(salary) OVER(PARTITION BY name ORDER BY name,month RANGE BETWEEN CURRENT ROW AND CURRENT ROW ) as range\_income

FROM employee

ORDER BY name, month



ROWS vs RANGE appears to cause much confusion.

ROWS means the specific row or rows specified, and RANGE refers to those same rows plus any others that have the same matching values.

ROWS | RANGE

|  |
| --- |
| **适用范围**：SQL Server 2012 到 SQL Server 2016。 |

通过指定分区中的起点和终点，进一步限制分区中的行数。 这是通过按照逻辑关联或物理关联对当前行指定某一范围的行实现的。 物理关联通过使用 ROWS 子句实现。

ROWS 子句通过指定当前行之前或之后的固定数目的行，限制分区中的行数。 此外，RANGE 子句通过指定针对当前行中的值的某一范围的值，从逻辑上限制分区中的行数。 基于 ORDER BY 子句中的顺序对之前和之后的行进行定义。 窗口框架“RANGE … CURRENT ROW …”包括在 ORDER BY 表达式中与当前行具有相同值的所有行。 例如，ROWS BETWEEN 2 PRECEDING AND CURRENT ROW 意味着该函数对其操作的行的窗口在大小上是 3 行、以当前行之前（包括当前行）的 2 行开头。

|  |
| --- |
| **System_CAPS_ICON_note.jpg 注意** |
| ROWS 或 RANGE 要求指定 ORDER BY 子句。 如果 ORDER BY 包含多个顺序表达式，则 CURRENT ROW FOR RANGE 在确定当前行时将考虑 ORDER BY 列表中的所有列。 |

UNBOUNDED PRECEDING

|  |
| --- |
| **适用范围**：SQL Server 2012 到 SQL Server 2016。 |

指定窗口在分区中的第一行开始。 UNBOUNDED PRECEDING 只能指定为窗口起点。

<无符号值指定> PRECEDING  
使用 <无符号值指定> 指示要置于当前行之前的行或值的数目。 对于 RANGE 则不允许这样指定。

CURRENT ROW

|  |
| --- |
| **适用范围**：SQL Server 2012 到 SQL Server 2016。 |

在与 ROWS 一起使用时指定窗口在当前行开始或结束，或者在与 RANGE 一起使用时指定当前值。 CURRENT ROW 可指定为既是起点，又是终点。

BETWEEN <窗口框架限定 > AND <窗口框架限定 >

|  |
| --- |
| **适用范围**：SQL Server 2012 到 SQL Server 2016。 |

与 ROWS 或 RANGE 一起使用，以便指定窗口的下（开始）边界和上（结束）边界点。 <窗口框架限定> 定义边界起���，<窗口框架限定> 定义边界终点。 上限不能小于下限。

UNBOUNDED FOLLOWING

|  |
| --- |
| **适用范围**：SQL Server 2012 到 SQL Server 2016。 |

指定窗口在分区的最后一行结束。 UNBOUNDED FOLLOWING 只能指定为窗口终点。 例如，RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING 定义以当前行开始、以分区的最后一行结束的窗口。

<无符号值指定> FOLLOWING  
使用 <无符号值指定> 指示要置于当前行之后的行或值的数目。 在 <无符号值指定> FOLLOWING 指定为窗口起点时，终点必须是 <无符号值指定>FOLLOWING。 例如，ROWS BETWEEN 2 FOLLOWING AND 10 FOLLOWING 定义一个窗口，该窗口以跟随在当前行之后的第二行开头、以跟随在当前行之后的第十行结尾。 对于 RANGE 则不允许这样指定。

无符号整数文字

|  |
| --- |
| **适用范围**：SQL Server 2012 到 SQL Server 2016。 |

一个正整数文字（包括 0），它指定要置于当前行或值之前或之后的行或值的数目。 这一指定仅对于 ROWS 有效。

## [一般备注](javascript:void(0))

可以在单个查询中将多个开窗函数与单个 FROM 子句一起使用。 每个函数的 OVER 子句在分区和排序上可能不同。

如果未指定 PARTITION BY，则此函数将查询结果集的所有行视为单个组。

如 果未指定 ORDER BY，则整个分区将用于窗口框架。 这仅适用于不要求 ORDER BY 子句的函数。 如果未指定 ROWS/RANGE，但指定了 ORDER BY，则将 RANGE UNBOUNDED PRECEDING AND CURRENT ROW 用作窗口框架的默认值。 这仅适用于可接受可选 ROWS/RANGE 指定的函数。 例如，排名函数无法接受 ROWS/RANGE，因此，此窗口框架不适用，甚至在存在 ORDER BY 而不存在 ROWS/RANGE 时也是如此。

如果指定 ROWS/RANGE 并且 <窗口框架前置> 用于 <窗口框架区>（简短语法），则这一指定用于窗口框架边界起点并且 CURRENT ROW 用于边界终点。 例如，“ROWS 5 PRECEDING”等于“ROWS BETWEEN 5 PRECEDING AND CURRENT ROW”。

## [限制和局限](javascript:void(0))

OVER 子句不能与 CHECKSUM 聚合函数结合使用。

RANGE 不能用于 <无符号值指定> PRECEDING 或 <无符号值指定> FOLLOWING。

根据用于 OVER 子句的排名、聚合或分析函数，可能不支持 <ORDER BY 子句> 和/或 <ROWS 和 RANGE 子句>。

* CASE .. WHEN .. THEN … ELSE xxx END

可以应用于 SELECT 也可以放在 WHERE 等其他

CASE categoryid

WHEN 1 THEN 'Beverages'

WHEN 2 THEN 'Condiments'

WHEN 3 THEN 'Confections'

WHEN 4 THEN 'Dairy Products'

WHEN 5 THEN 'Grains/Cereals'

WHEN 6 THEN 'Meat/Poultry'

WHEN 7 THEN 'Produce'

WHEN 8 THEN 'Seafood'

ELSE 'Unknown Category'

END AS categoryname

SELECT

CASE

WHEN phone is null THEN 'aaaa'

ELSE phone

END,

count(0)

FROM acct\_users

GROUP BY

CASE

WHEN phone is null THEN 'aaaa'

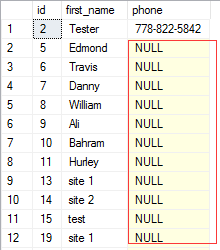
ELSE phone

END

* NULL value

在条件里 null 视同于 false

SELECT id, first\_name, phone FROM acct\_users

WHERE phone not like '%604%'



SELECT id, first\_name, phone FROM acct\_users

WHERE phone not like '%604%' OR phone is null

//注意：不能是 phone = null; phone is null | not null

容易产生 null 的情况： LEFT JOIN, RIGHT JOIN 等

SELECT

a.id,

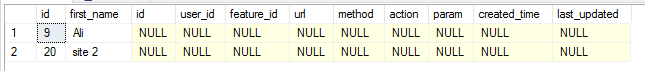
a.first\_name,

b.\*

FROM acct\_users a

LEFT JOIN acct\_user\_audit b ON (a.id = b.user\_id)

WHERE b.user\_id is null



解决方案：

1. 使用 OR colname is null
2. ISNULL ( a, b)

说明：如果a和b同时为NULL，返回NULL，如果a为NULL，b不为NULL，返回b，如果a不为NULL，b为NULL返回a，如果a和b都不为NULL返回a

1. IIF(col, true\_value, false\_value) IIF(b.user\_id is null, 999, 111);
2. NULLIF(a, b);
3. ISNUMERIC(val) - 判断是否数值型， 不是则返回 0 ， 是则返回 1 ， 注意不是数值本身

‘4a’ – 0 ‘2012-12-25’ - 0

说明：NULLIF(a, b) 简单说 a=b 则返回null, 否则返回 a (a 本身也有可能是null)

如果 a 和 b 相等， 返回 null , 注意 123 = ‘123’

如果 a 和 b 不相等， 返回 a 不管 a 是否是 null, 如果 a is null, 返回 null

如果a和b同时为NULL，返回NULL，

SELECT

a.id,

a.first\_name,

ISNULL(b.user\_id, -1) AS uid,

b.\*

FROM acct\_users a

LEFT JOIN acct\_user\_audit b ON (a.id = b.user\_id)

WHERE ISNULL(b.user\_id, -1) = -1

* 数据类型

**Syntax**

CAST(*value* AS *datatype*)

TRY\_CAST(*value* AS *datatype*)

CONVERT (*datatype*, *value* [, *style\_number*])

TRY\_CONVERT (*datatype*, *value* [, *style\_number*])

PARSE (*value* AS *datatype* [USING *culture*])

TRY\_PARSE (*value* AS *datatype* [USING *culture*])

* 日期类型

TODATETIMEOFFSET(*date\_and\_time\_value*, *time\_zone*)

SWITCHOFFSET(*datetimeoffset\_value, time\_zone*)

EOMONTH(*input* [, *months\_to\_add*]) – 返回月底最后一天， 返回类型是DATE

SELECT

DATEFROMPARTS(2012, 02, 12),

DATETIME2FROMPARTS(2012, 02, 12, 13, 30, 5, 1, 7),

DATETIMEFROMPARTS(2012, 02, 12, 13, 30, 5, 997),

DATETIMEOFFSETFROMPARTS(2012, 02, 12, 13, 30, 5, 1, -8, 0, 7),

SMALLDATETIMEFROMPARTS(2012, 02, 12, 13, 30),

TIMEFROMPARTS(13, 30, 5, 1, 7);

* JOINS

CROSS JOIN : it will generate Cartesian result set, the number of rows in the first table multiply by the number of rows in the second table

INNER JOIN (JOIN) – Matching the records base on predicate expression

OUTER JOIN : LEFT JOIN , RIGHT JION , FULL JOIN

LEFT JOIN – the rows of the left table are preserved; it will use NULL value to fill the nonpreserved rows in the right table.

FULL JOIN – keep both sides of tables preserve the rows. It will use null value to fill the nonpreserved rows in both sides of tables.

JOIN 可以应用到 CRUD 操作上

INSERT mmlog (ref\_id, subsys, num1)

SELECT b.id, b.name, 10

FROM mmlog a

INNER JOIN mmsys b on (a.ref\_id = b.id)

WHERE b.id = 1

DELETE FROM a

FROM mmlog a

INNER JOIN mmsys b on (a.ref\_id = b.id)

WHERE a.subsys = 'Sholve'

UPDATE a SET a.subsys = concat(a.subsys, ' ' , b.id)

FROM mmlog a

INNER JOIN mmsys b on (a.ref\_id = b.id)

WHERE b.id = 1

SELECT

ref\_id,

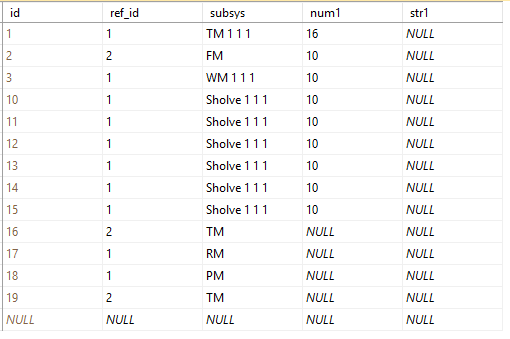
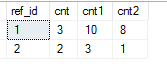
COUNT(DISTINCT ISNULL(num1,0) ) AS cnt,

COUNT(ISNULL(num1,0)) AS cnt1,

COUNT(num1) AS cnt2

FROM mmlog

GROUP BY ref\_id

* CTE

WITH employee(id, name, title, mgrname, mgrtitle) AS (

SELECT a.id, a.name, a.title, b.name, b.title

FROM emp a

LEFT JOIN emp b ON (a.id = b.mgr\_id)

)

SELECT \* FROM employee;

WITH cte\_table( new\_colname, new\_colname1 , ….) as (

SELECT … FROM …

)

可以重新对字段命名， 字段个数和选择的字段个数必须完全相同

WITH employee AS (

SELECT a.id, a.name, a.title, b.name as mname, b.title as mtitle

FROM emp a

LEFT JOIN emp b ON (a.id = b.mgr\_id)

)

SELECT \* FROM employee;

也可以不指定， 在SELECT 语句里重新命名， 尤其是名称相同时

定义多重

WITH t1 AS (

SELECT id, name, title, mgr\_id

FROM emp

), manager AS (

SELECT a.id, a.name, a.title, b.name AS mgr, b.title AS mgrtitle

FROM t1 a

LEFT JOIN emp b ON (a.mgr\_id = b.id)

)

SELECT \* FROM manager;

WITH lwh as

(

SELECT

ref\_id,

COUNT(DISTINCT ISNULL(num1,0) ) AS cnt,

COUNT(ISNULL(num1,0)) AS cnt1,

COUNT(num1) AS cnt2

FROM mmlog

GROUP BY ref\_id

), will as (

SELECT \* FROM mmlog

WHERE ref\_id = 2

)

SELECT \* FROM will;

SELECT \* FROM lwh; -- 执行第二个语句时出错， 因为只能是紧跟的语句有效

由于表的表达式，只能在接下来的语句里被调用一次， 所以多重表表达式， 前面的表达式是给后面的表达式使用的， 否则无关联， 前面的就没有意义了

* Recursive CTE – 迭代表表达式， 用于树形结构很合适

WITH t1 AS (

SELECT mgr\_id as manager\_id, id as empid, name, title

FROM emp

WHERE mgr\_id = 0 -- manage id 从 0 开始

UNION ALL -- 用上面的 t1 表作为员工表， 联合此员工下面的所有下属

SELECT

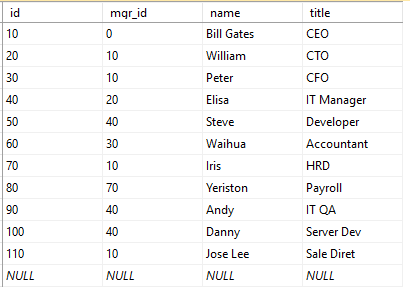
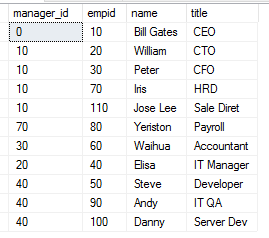
b.mgr\_id as manager\_id, b.id as empid, b.name, b.title

FROM t1 a

INNER JOIN emp b ON ( a.empid = b.mgr\_id)

)

SELECT \* FROM t1;

上面的表达式，实际上没有什么意义， 完全可以用下面语句代替

SELECT mgr\_id as manager\_id, id as empid, name, title from emp

ORDER BY mgr\_id, id

WITH t1 AS (

SELECT

1 as grade,

mgr\_id as manager\_id, id as empid, name, title

FROM emp

WHERE mgr\_id = 0

UNION ALL

SELECT

a.grade + 1 as grade,

b.mgr\_id as manager\_id, b.id as empid, b.name, b.title

FROM t1 a

INNER JOIN emp b ON ( a.empid = b.mgr\_id)

)

SELECT \* FROM t1;

可以增加一些逻辑运算， 如计算递归的级数等

Notes: DERIVE TABLE and CTE are both in Memory Object, Once it have been executed , it will gone.

Derive table : SELECT \* FROM ( select \* from xxx ) WHERE ….

Derive talbe: is data source , is recordset

SubQuery: - use for WHERE clause to filter

* VIEW and inline table-value function ( TVF )

IF OBJECT\_ID('Sales.USACusts') IS NOT NULL

DROP VIEW Sales.USACusts;

GO

CREATE VIEW Sales.USACusts

AS

SELECT

custid, companyname, contactname, contacttitle, address,

city, region, postalcode, country, phone, fax

FROM Sales.Customers

WHERE country = N'USA';

GO

CREATE VIEW vw\_emp AS

SELECT \* FROM emp ORDER BY id desc

The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries, and common table expressions, unless TOP, OFFSET or FOR XML is also specified.

在 VIEW 里面指定 ORDER BY 是不允许的， 除非有 TOP, OFFSET .. FETCH NEXT… ONLY , FOR XML

CREATE VIEW vw\_emp1 AS

SELECT TOP 10 \* FROM emp ORDER BY id desc

这个是允许的，因为有 TOP 10

WITH SCHEMABINDING – 不允许修改关联的对象， 如表， 修改表等操作，会出错

CREATE VIEW xxxxx WITH SCHEMABINDING AS

ALTER VIEW Sales.USACusts WITH SCHEMABINDING

AS

SELECT

custid, companyname, contactname, contacttitle, address,

city, region, postalcode, country, phone, fax

FROM Sales.Customers

WHERE country = N'USA'

WITH CHECK OPTION;

GO

* CROSS JOIN vs CROSS APPLY

SELECT \* FROM mmsys a

CROSS APPLY ( SELECT \* FROM mmlog b WHERE a.id = b.ref\_id ) c

SELECT \* FROM mmsys a

CROSS JOIN ( SELECT \* FROM mmlog b WHERE a.id = b.ref\_id ) c

// 出错， 并传递行数据

* LEFT OUTER JOIN vs OUTER APPLY

APPLY 的意思是， 左边表的每一行数据都执行apply, 也就是把每行数据参数传入 apply 里可以作为过滤条件使用

SELECT \* FROM mmsys a

LEFT OUTER JOIN ( SELECT \* FROM mmlog b) c ON (a.id = c.ref\_id )

SELECT \* FROM mmsys a

OUTER APPLY ( SELECT \* FROM mmlog b) c ON (a.id = c.ref\_id )

// 出错， 不允许有 ON (…)

SELECT \* FROM mmsys a

OUTER APPLY ( SELECT \* FROM mmlog b WHERE a.id = b.ref\_id ) c

// 把行数据作为参数传递

* **Inline Table-Valued Functions**

RETURNS SCALAR VALUE

CREATE FUNCTION f3(@id INT)

RETURNS INT

AS

BEGIN

DECLARE @maxid int;

SELECT @maxid = MAX(cust\_id) FROM Orders WHERE id >= @id;

RETURN @maxid;

END

CREATE FUNCTION f4(@id INT)

RETURNS INT

AS

RETURN SELECT MAX(cust\_id) FROM Orders WHERE id >= @id;

// 这个出错： 因为返回的其实是数据集

CREATE FUNCTION f4 (@id INT)

RETURNS TABLE

AS

RETURN SELECT MAX(cust\_id) AS mid FROM Orders WHERE id >= @id;

// 这样就不会出错

RETURNS INLINE TABLE VALUE

CREATE FUNCTION f5(@id int)

RETURNS TABLE

AS

RETURN SELECT id, cust\_id, amount FROM Orders WHERE cust\_id = @id;

/ / 注意：RETURNS TABLE AS 后面只能使用 RETURN

此时可以体现出 OUTER APPLY , CROSS APPLY 的优越性了

SELECT \* FROM customer a

CROSS APPLY f5(a.id) -- 相当于 INNER JOIN

SELECT \* FROM customer a

OUTER APPLY f5(a.id) -- 相当于 LEFT JOIN

SELECT \* FROM customer a

LEFT JOIN f5(a.id) b ON (a.id = b.cust\_id)

// f5(a.id) 出错，不能应用表 a.id

SELECT \* FROM customer a

LEFT JOIN f5(2) b ON (a.id = b.cust\_id)

// 可以使用固定的值 ： f5(2) 意义就不大了

USE TSQL2012;

IF OBJECT\_ID('dbo.GetCustOrders') IS NOT NULL

DROP FUNCTION dbo.GetCustOrders;

GO

CREATE FUNCTION dbo.GetCustOrders

(@cid AS INT) RETURNS TABLE

AS

RETURN

SELECT orderid, custid, empid, orderdate, requireddate,

shippeddate, shipperid, freight, shipname, shipaddress, shipcity,

shipregion, shippostalcode, shipcountry

FROM Sales.Orders

WHERE custid = @cid;

GO

自定义表结构：

CREATE FUNCTION f6(@minid int, @maxid int)

RETURNS @tab table (

id int primary key not null identity(10,10),

oid int,

ddd datetime,

num1 int

)

AS

BEGIN

DECLARE @sid int;

set @sid = 100;

WITH t1 AS (

SELECT id, odate, amount

FROM orders

WHERE id >= @minid AND id <= @maxid AND id <= @sid

)

INSERT @tab(oid, ddd, num1) SELECT \* FROM t1;

RETURN;

END

重点：

1. RETURNS @tab TABLE （ …. ）
2. 必须对表变量 @tab 赋值
3. 必须有 RETURN , 并且不带任何参数

FUNCTION : 不允许带 output 参数， 所有参数都是input

CREATE FUNCTION f8(@id int, @maxid int)

RETURNS INT

AS

BEGIN

DECLARE @out int;

SELECT @out=MAX(id) FROM orders

set @maxid = @out;

return @out;

END

DECLARE @m int;

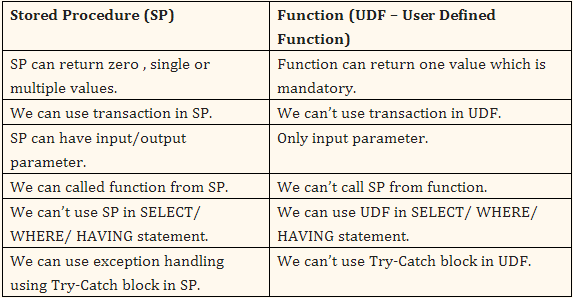
set @m = 100;

SELECT dbo.f8(1, @m);

print @m; -- 打印输出仍然是 100， 并没有被改变

STORE PROCEDURE vs FUNCTION :

1. If Procedure using Return , only return Scalar Integer value, FUNCTION require Return (scalar value, inline table value)
2. Store Proceduce can be executed independently , FUNCTION must be used in SQL Express Statement.
3. Store Proceduce support input/output parameter, FUNCTION only input parameter



* UNION ALL , UNION

SELECT lname FROM Customer

UNION ALL

SELECT name FROM Employee

ORDER BY lname

存在重复行， ORDER BY 对最后的结果集排序， 所以字段名应该是第一个表

SELECT lname FROM Customer

UNION

SELECT name FROM Employee

ORDER BY lname

没有重复行， 相当于对最后结果结合使用 distinct

* INTERSECT , INTERSECT ALL (没有)

SELECT lname FROM Customer

INTERSECT

SELECT name FROM Employee

ORDER BY lname

// 没有重复行， 相当于对最后结果集合进行 distinct

INTERSECT ALL 的解决方案：

WITH INTERSECT\_ALL AS

(

SELECT

ROW\_NUMBER() OVER(PARTITION BY lname ORDER BY (SELECT 0)) AS rid,

lname

FROM customer

INTERSECT

SELECT

ROW\_NUMBER() OVER(PARTITION BY name ORDER BY (SELECT 0)) AS rid,

name

FROM Employee

)

SELECT \* FROM INTERSECT\_ALL

* EXCEPT, EXCEPT ALL (没有)

SELECT name FROM Employee

EXCEPT

SELECT lname FROM customer

去除重复的行，并且对最后结果集执行 distinct

EXCEPT ALL 的解决方案：

WITH EXCEPT\_ALL AS

(

SELECT

ROW\_NUMBER() OVER(PARTITION BY name ORDER BY (SELECT 0)) AS rid,

name

FROM Employee

EXCEPT

SELECT

ROW\_NUMBER() OVER(PARTITION BY lname ORDER BY (SELECT 0)) AS rid,

lname

FROM customer

)

SELECT \* FROM EXCEPT\_ALL

Precedence:

*INTERSECT* operator precedes *UNION* and *EXCEPT*, and *UNION* and *EXCEPT* are considered equal. INTERSECT 优先， 其次是 UNION 和 EXCEPT (他们同级)

To control the order of evaluation of set operators, use parentheses, because they have the highest

precedence. Also, using parentheses increases the readability, thus reducing the chance for errors.

(SELECT country, region, city FROM Production.Suppliers

EXCEPT

SELECT country, region, city FROM HR.Employees)

INTERSECT

SELECT country, region, city FROM Sales.Customers;

* ROW\_NUMBER(), RANK(), DENSE\_RANK()

SELECT

id,

name,

salary,

ROW\_NUMBER() OVER(PARTITION BY name ORDER BY name, salary) as rowno,

RANK() OVER(PARTITION BY name ORDER BY name, salary) as rankno,

DENSE\_RANK() OVER(PARTITION BY name ORDER BY name, salary) as drankno

FROM Employee

RANK() – There is rank number gap 1, 2, 3, 3, 5 gap 4

DENSE\_RANK() – There is no gap in ranking, 1, 2, 3, 3, 4, 5



* OFFSET : LAG(), LEAD(), FIRST\_VALUE(), LAST\_VALUE()

The *LAG* function looks before the current row, and the *LEAD* function looks ahead ( after current row ).

LAG( colname [, offset [ , default\_value] ] ) offset default 1

SELECT

id,

name,

salary,

ROW\_NUMBER() OVER(PARTITION BY name ORDER BY name, salary) as rowno,

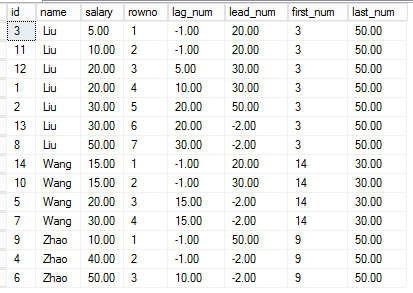
LAG(salary, 2, -1) OVER(PARTITION BY name ORDER BY name, salary) as lag\_num,

LEAD(salary, 2, -2) OVER(PARTITION BY name ORDER BY name, salary) as lead\_num,

FIRST\_VALUE(id) OVER(PARTITION BY name ORDER BY name) as first\_num,

LAST\_VALUE(salary) OVER(PARTITION BY name ORDER BY name) as last\_num

FROM Employee



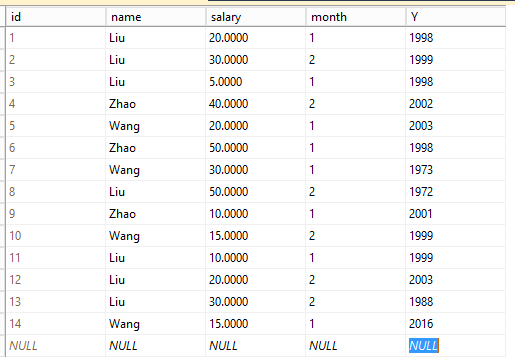
* PIVOT TABLE , UNPIVOT TABLE

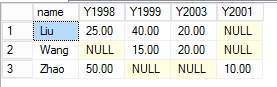
SELECT name, [1998] AS Y1998, [1999] AS Y1999, [2003] AS Y2003, [2001] AS Y2001

FROM

(SELECT name, salary, Y FROM Employee) a

PIVOT( SUM(salary) FOR Y IN ([1998],[1999],[2003],[2001]) ) b





PIVOT( SUM(salary) FOR Y IN ([1998],[1999],[2003],[2001]) ) b

[1998] 是 Y 的值， 但是也是结果集合的字段名

SELECT name, [1998] AS Y1998 – 最好通过别名更改

所以如何动态生成字段， 这就需要自己写代码

UNPIVOT TABLE

SELECT name, Y, income FROM test2

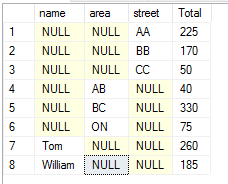
UNPIVOT(income FOR Y IN(Y1998, Y1999, Y2003, Y2001) ) p

* GROUPING SETS , CUBE, ROLLUP, GROUPING\_ID

GROUPING SETS - The *GROUPING SETS* subclause is a powerful enhancement to the *GROUP BY* clause that is used mainly in

reporting and data warehousing. By using this subclause, you can define multiple grouping sets in the same

query.

multiple grouping set combination in the same query. Combination is flexible defined

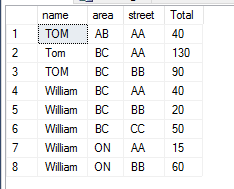
SELECT name, area, street, sum(num) as Total

FROM orders

GROUP BY

GROUPING SETS

(name,area,street) - 单独对 name, area, street 做聚集

SELECT name, area, street, sum(num) as Total

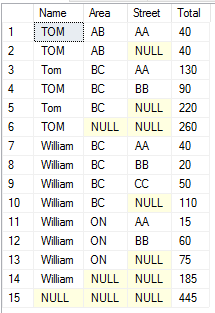
FROM orders

GROUP BY

GROUPING SETS (

(name,area,street) - 注意区别：对（name,area,street）组合进行聚合

)



SELECT Name, Area, Street, sum(num) as Total

FROM orders

GROUP BY

GROUPING SETS (

(name,area,street), - 各层都可以聚集汇总

(name,area), - 各层汇总组合可以自由组合，

(name), - 但是如果随意组合逻辑混乱

()

)

CUBE:

The *CUBE* subclause of the *GROUP BY* clause provides an abbreviated way to define multiple grouping sets.

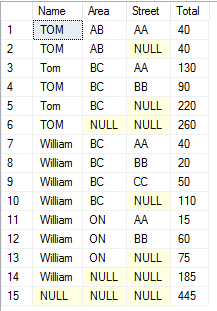
*CUBE(a, b, c)* is equivalent to *GROUPING SETS( (a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), () )*

SELECT Name, Area, Street, sum(num) as Total

FROM orders

GROUP BY

CUBE(name, area, street)



ROLLUP:

*ROLLUP(a, b, c) - GROUPING SETS( (a, b, c), (a, b), (a), () )*.

SELECT Name, Area, Street, sum(num) as Total

FROM orders

GROUP BY

ROLLUP (name, area, street)

GROUPING\_ID :

GROUPING\_ID *(a, b, c, d) -* (0×8 + 0×4 + 0×2 + 0×1). The grouping set *(a, c)* is represented by the integer 5 (0×8 + 1×4 + 0×2 +

1×1), and so on.

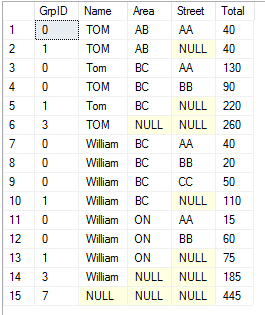
rely on the *NULL, if null value just put 1 in the position , calculate the result*

SELECT GROUPING\_ID(name,area,street) AS GrpID, Name, Area, Street, sum(num) as Total

FROM orders

GROUP BY

ROLLUP (name, area, street)



* DATA Modification

Remember that both *@@identity* and *SCOPE\_IDENTITY* return the last identity value produced by

the current session. Neither is affected by inserts issued by other sessions. However, if you want to

know the current identity value in a table (the last value produced) regardless of session, you should

use the *IDENT\_CURRENT* function and provide the table name as input.

Both *@@identity* and *SCOPE\_IDENTITY* returned *NULL* marks because no identity values were created

in the session in which this query ran. *IDENT\_CURRENT* returned the value 4 because it returns

the current identity value in the table, regardless of the session in which it was produced.

* INSERT

CREATE PROCEDURE f15

AS

SELECT name FROM emp;

SELECT name FROM orders; //如果是字段不匹配，则出错 SELECT name, area FROM orders;

insert test1(name) exec f15

可以将两表的内容插入

* DELETE vs TRUNCATE

TRUNCATE – can not truncate the table which has foreign key, TRUNCATE will reset IDENTITY column

Truncate it remove data directly without copy it to the rollback log.

Truncate much faster

* Merging Data

[ WITH <common\_table\_expression> [,...n] ]

MERGE

[ TOP ( expression ) [ PERCENT ] ]

[ INTO ] <target\_table> [ WITH ( <merge\_hint> ) ] [ [ AS ] table\_alias ]

USING <table\_source>

ON <merge\_search\_condition>

[ WHEN MATCHED [ AND <clause\_search\_condition> ]

THEN <merge\_matched> ] [ ...n ]

[ WHEN NOT MATCHED [ BY TARGET ] [ AND <clause\_search\_condition> ]

THEN <merge\_not\_matched> ]

[ WHEN NOT MATCHED BY SOURCE [ AND <clause\_search\_condition> ]

THEN <merge\_matched> ] [ ...n ]

[ <output\_clause> ]

[ OPTION ( <query\_hint> [ ,...n ] ) ]

;

<target\_table> ::=

{

[ database\_name . schema\_name . | schema\_name . ]

target\_table

}

<merge\_hint>::=

{

{ [ <table\_hint\_limited> [ ,...n ] ]

[ [ , ] INDEX ( index\_val [ ,...n ] ) ] }

}

<table\_source> ::=

{

table\_or\_view\_name [ [ AS ] table\_alias ] [ <tablesample\_clause> ]

[ WITH ( table\_hint [ [ , ]...n ] ) ]

| rowset\_function [ [ AS ] table\_alias ]

[ ( bulk\_column\_alias [ ,...n ] ) ]

| user\_defined\_function [ [ AS ] table\_alias ]

| OPENXML <openxml\_clause>

| derived\_table [ AS ] table\_alias [ ( column\_alias [ ,...n ] ) ]

| <joined\_table>

| <pivoted\_table>

| <unpivoted\_table>

}

<merge\_search\_condition> ::=

<search\_condition>

<merge\_matched>::=

{ UPDATE SET <set\_clause> | DELETE }

<set\_clause>::=

SET

{ column\_name = { expression | DEFAULT | NULL }

| { udt\_column\_name.{ { property\_name = expression

| field\_name = expression }

| method\_name ( argument [ ,...n ] ) }

}

| column\_name { .WRITE ( expression , @Offset , @Length ) }

| @variable = expression

| @variable = column = expression

| column\_name { += | -= | \*= | /= | %= | &= | ^= | |= } expression

| @variable { += | -= | \*= | /= | %= | &= | ^= | |= } expression

| @variable = column { += | -= | \*= | /= | %= | &= | ^= | |= } expression

} [ ,...n ]

<merge\_not\_matched>::=

{

INSERT [ ( column\_list ) ]

{ VALUES ( values\_list )

| DEFAULT VALUES }

}

<clause\_search\_condition> ::=

<search\_condition>

<search condition> ::=

{ [ NOT ] <predicate> | ( <search\_condition> ) }

[ { AND | OR } [ NOT ] { <predicate> | ( <search\_condition> ) } ]

[ ,...n ]

<predicate> ::=

{ expression { = | < > | ! = | > | > = | ! > | < | < = | ! < } expression

| string\_expression [ NOT ] LIKE string\_expression

[ ESCAPE 'escape\_character' ]

| expression [ NOT ] BETWEEN expression AND expression

| expression IS [ NOT ] NULL

| CONTAINS

( { column | \* } , '< contains\_search\_condition >' )

| FREETEXT ( { column | \* } , 'freetext\_string' )

| expression [ NOT ] IN ( subquery | expression [ ,...n ] )

| expression { = | < > | ! = | > | > = | ! > | < | < = | ! < }

{ ALL | SOME | ANY} ( subquery )

| EXISTS ( subquery ) }

<output\_clause>::=

{

[ OUTPUT <dml\_select\_list> INTO { @table\_variable | output\_table }

[ (column\_list) ] ]

[ OUTPUT <dml\_select\_list> ]

}

<dml\_select\_list>::=

{ <column\_name> | scalar\_expression }

[ [AS] column\_alias\_identifier ] [ ,...n ]

<column\_name> ::=

{ DELETED | INSERTED | from\_table\_name } . { \* | column\_name }

| $action

MERGE INTO Customer

USING VIP

ON Customer.id = VIP.id -- 条件

WHEN MATCHED AND (Customer.name = VIP.vname) THEN -- AND Customer.name = VIP.vname

UPDATE SET

salary = salary + VIP.income,

address = 'U:' + VIP.addr,

comment = 'Matched UPDATE'

MERGE INTO Customer

USING VIP

ON Customer.id = VIP.id

WHEN MATCHED AND (LEFT(Customer.name,1) = 'W' OR LEFT(vname,1) = 'L') THEN

DELETE;

注意：

1. WHEN MATCHED 只能最多有一个， 也可以没有
2. WHEN MATCHED AND 条件可以是 TARGET的字段 或者 SOURCE 的字段
3. WHEN MATCHED 可以执行 UPDATE SET | DELETE 操作

MERGE INTO Customer

USING VIP

ON Customer.id = VIP.id

WHEN NOT MATCHED THEN

INSERT(id, name, salary, address, comment)

values(id, vname, income, addr, 'Inserted');

MERGE INTO Customer

USING VIP

ON Customer.id = VIP.id

WHEN NOT MATCHED BY TARGET AND LEFT(vname,1) <> 'W' THEN - AND 条件判断只能是用源表字段

INSERT(id, name, salary, address, comment) - 只能使用 INSERT

values(id, vname, income, addr, 'Inserted');

注意：

1）NOT MATCHED 默认是 BY TARGET - 也就是说在目标表找不到该记录, 如果要附带 AND 条件:

2) 必须是源表的字段， 不能是目标表的字段

3) NOT MATCHED BY TARGET 只能使用 INSERT(…) VALUES (…) 语句

4) NOT MATCHED BY TARGET 最多只能有一个，也可以没有

MERGE INTO Customer

USING VIP

ON Customer.id = VIP.id

WHEN NOT MATCHED BY SOURCE THEN

UPDATE SET

salary = -1000,

comment = 'DELETED';

MERGE INTO Customer

USING VIP

ON Customer.id = VIP.id

WHEN NOT MATCHED BY SOURCE AND LEFT(Customer.name,1) = 'C' THEN -- 必须是目标表上的字段

DELETE;

注意：

1）NOT MATCHED BY SOURCE - 也就是说在目标表有该记录, 源表上没有

2）必须是目标表上的字段， 不能是源表上的字段

3）NOT MATCHED BY SOURCE 只能使用 UPDATE SET | DELETE

4) NOT MATCHED BY SOURCE 最多只能有一个，也可以没有

OUTPUT:

MERGE INTO Customer

USING VIP

ON Customer.id = VIP.id

WHEN MATCHED THEN

UPDATE SET

address = addr,

salary = income,

comment = 'Matched Update'

WHEN NOT MATCHED THEN

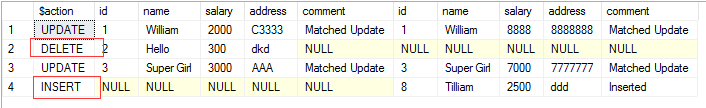
INSERT(id,name, salary, address, comment)

values(id, vname, income, addr, 'Inserted')

WHEN NOT MATCHED BY SOURCE THEN

DELETE

OUTPUT $action, deleted.\*, inserted.\*;



MERGE TOP(10) INTO Customer - 只执行前 10 个记录

* CTE – Modify DATA

WITH t1 AS (

SELECT a.id as cid, a.name as cname, b.id as oid, b.area, b.num

FROM CUSTOMER a

LEFT JOIN ORDERS b ON (a.id = b.cust\_id)

)

UPDATE t1 SET cname = cname + ' z', num = num + 2;

select \* from Customer;

select \* from Orders;

注意：

1） 以上出错， CTE 不能同时更改两个表， 只能操作一个表的字段

INSERT t1(cid, cname) values(20, 'Tommy'); -- 只要操作一个表就可以

DELETE Data:

DELETE FROM Customer ORDER BY id OFFSET 3 ROW FETCH NEXT 2 ROW ONLY;

* 如果这样删除数据是不允许的， 必须借助 CTE

UPDATE Orders SET num += 100 ORDER BY id OFFSET 5 ROW FETCH NEXT 2 ROW ONLY;

* 这样更新是不可以的

DELETE TOP(2) FROM Customer - 删除顶上 2 个记录，是可以的

UPDATE TOP(3) Orders SET num += 100; 也是可以的

WITH T1 AS (

SELECT \* FROM Customer

ORDER BY id DESC

OFFSET 3 ROW FETCH NEXT 2 ROW ONLY

)

DELETE FROM t1;

* 这样就可以删除了 2 条记录

WITH t1 as (

SELECT \* FROM Orders

ORDER BY id

OFFSET 5 ROW FETCH NEXT 2 ROW ONLY

)

UPDATE T1 SET num += 200;

* 这样就可以更新了 2 条记录

UPDATE Orders SET num += 20

OUTPUT deleted.num, inserted.num; - both deleted and inserted

DELETE Orders

OUTPUT deleted.num - only deleted

WHERE id >= 10;

INSERT Orders (name, area, street)

OUTPUT inserted.\* - only inserted

values('Sodongpo', 'Guangzhou', 'Yanghc');

* OUTPUT 表可以放在子查询里， 被再次利用

INSERT INTO dbo.ProductsAudit(productid, colname, oldval, newval)

SELECT productid, N'unitprice', oldval, newval

FROM (

UPDATE dbo.Product SET unitprice \*= 1.15

OUTPUT

inserted.productid,

deleted.unitprice AS oldval,

inserted.unitprice AS newval

WHERE supplierid = 1

) AS D

WHERE oldval < 20.0 AND newval >= 20.0;

**数据库快照**

数据库快照是数据库某个时间点的一个只读源数据的静态视图。 快照创建时只是创建了一个空的数据文件。 只有当源数据页面发生更改时， 才会将更改前的旧页面数据复制到快照的页面上。如果源数据没有更改， 则仍然使用源数据的页面数据。

CREATE DATABASE \*database\_snapshot\_name\*

ON

(

NAME =\*logical\_file\_name\*,

FILENAME ='\*os\_file\_name\*'

) [ ,...\*n\* ]

AS SNAPSHOT OF \*source\_database\_name\*

[;]

CREATE DATABASE LWH\_SNAP0721 ON (

NAME=LWH, - 源数据的

FILENAME = 'd:\temp\LWH\_SNAP0721.snap'

)

AS SNAPSHOT OF LWH

如果遇到有多个逻辑文件的情况：

CREATE DATABASE LWH\_SNAP0721 ON (

NAME=LWH1,

FILENAME = 'd:\temp\LWH\_SNAP072101.snap'

),

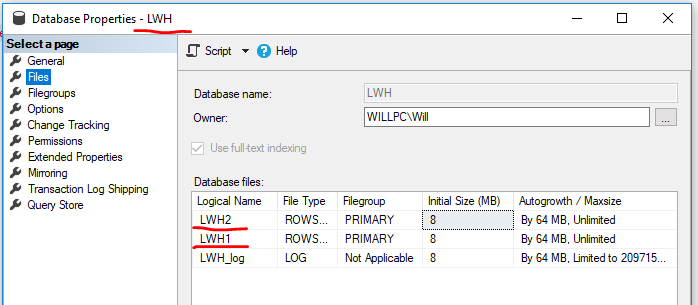
(

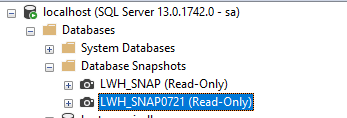
NAME=LWH2,

FILENAME = 'd:\temp\LWH\_SNAP070202.snap'

)

AS SNAPSHOT OF LWH





使用快照，就如同使用数据库一样。

USE LWH\_SNAP0721

SELECT \* FROM Student

# <https://blog.csdn.net/snowfoxmonitor/article/details/49557051>

# DBCC CHECKDB 数据库或表修复

2015年11月01日 11:47:29 [DoveFeng](https://me.csdn.net/snowfoxmonitor) 阅读数 7129

MS Sql Server 提供了很多数据库修复的命令，当数据库质疑或是有的无法完成读取时可以尝试这些修复命令。

　　1. DBCC CHECKDB   
　　重启服务器后，在没有进行任何操作的情况下，在SQL查询分析器中执行以下SQL进行数据库的修复，修复数据库存在的一致性错误与分配错误。

use master   
declare @databasename varchar(255)   
set @databasename='需要修复的数据库实体的名称'   
exec sp\_dboption @databasename, N'single', N'true' --将目标数据库置为单用户状态   
dbcc checkdb(@databasename,REPAIR\_ALLOW\_DATA\_LOSS)   
dbcc checkdb(@databasename,REPAIR\_REBUILD)   
exec sp\_dboption @databasename, N'single', N'false'--将目标数据库置为多用户状态   
  
然后执行 DBCC CHECKDB('需要修复的数据库实体的名称') 检查数据库是否仍旧存在错误。注意：修复后可能会造成部分数据的丢失。   
  
2. DBCC CHECKTABLE   
如果DBCC CHECKDB 检查仍旧存在错误，可以使用DBCC CHECKTABLE来修复。   
use 需要修复的数据库实体的名称   
declare @dbname varchar(255)   
set @dbname='需要修复的数据库实体的名称'   
exec sp\_dboption @dbname,'single user','true'   
dbcc checktable('需要修复的数据表的名称',REPAIR\_ALLOW\_DATA\_LOSS)   
dbcc checktable('需要修复的数据表的名称',REPAIR\_REBUILD)   
------把’ 需要修复的数据表的名称’更改为执行DBCC CHECKDB时报错的数据表的名称   
exec sp\_dboption @dbname,'single user','false'   
  
3. 其他的一些常用的修复命令   
DBCC DBREINDEX 重建指定数据库中表的一个或多个索引   
用法：DBCC DBREINDEX (表名,’’) 修复此表所有的索引。   
  
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

SQL SERVER数据库的检测及修复方法   
随着K/3产品的推广，要求客户服务人员对SQL SERVER数据库的了解也进一步提高。在K/3的使用过程中，数据库文件被频繁地使用，由于某些原因，数据库有可能被损坏，本文将针对这种情况的数据库检测及修复方法做一简单讲解。希望各位在实际工作过程中有新的发现时，及时给我们提供信息，以便做进一步的更新。   
1.1 SQL SERVER数据库的检测   
SQL SERVER提供了数据库检测的命令，可用DBCC CHECKDB对数据库中各个对象的分配及结构的正确性进行检测，并可通过一参数控制，将所有的错误信息显示出来。其语法如下：   
DBCC CHECKDB   
('database\_name' [,NOINDEX | { REPAIR\_ALLOW\_DATA\_LOSS   
| REPAIR\_FAST   
| REPAIR\_REBUILD   
}]   
) [WITH {ALL\_ERRORMSGS | NO\_INFOMSGS}]   
参数说明：   
'database\_name'代表被检测的数据库实体名；   
NOINDEX指非系统表的非聚族索引不检测；   
REPAIR\_ALLOW\_DATA\_LOSS | REPAIR\_FAST| REPAIR\_REBUILD 指直接修复发现的错误，其中REPAIR\_ALLOW\_DATA\_LOSS代表，若此错误不能修复时，系统将直接删除相关数据。带此三个参数的任一个时，数据库必须处于单用户模式，可在Enterprise Manager中的数据库属性中设置；   
ALL\_ERRORMSGS代表将检测到的错误信息全部显示出来，否则，对于每张表最多只显示200条错误信息；   
NO\_INFOMSGS代表隐藏所有的信息及占用空间的报告。   
经过检测，对于错误的对象，将以OBJECT ID的形式报告具体出错的信息，可根据OBJECT ID到系统表sysobjects中查找到相关的表，即NAME。   
  
1.2 SQL SERVER问题数据库的修复   
经过数据库检测后，可针对出现的问题采取相应的措施进行处理。如通过检测后，发现对象的物理存放存在问题，可用DBCC CHECKALLOC来进行修复：   
DBCC CHECKALLOC ('database\_name' | REPAIR\_REBUILD }] ) [WITH {ALL\_ERRORMSGS | NO\_INFOMSGS}]   
若是非系统对象的索引出错，则可用DBCC DBREINDEX进行修复：   
DBCC DBREINDEX ( [ 'database.owner.table\_name' [, index\_name [, fillfactor ] ] ] ) [WITH NO\_INFOMSGS]   
以上两种情况，也可直接使用DBCC CHECKDB(‘db\_name’,repair\_rebuild)来修复。   
另外一种情况是在进行检测时，提示无法建立数据连接，此时表明，数据库已损坏。对于这种情况，我们可采取如下措施来尝试修复。   
首先，在SQL Enterprise中新建一数据库（如数据库名为test）,建好数据库后，停止SQL Server Service Manager，并将客户数据库的MDF文件更名为test \_data.mdf(即新建数据库的主文件名)，然后用更名后的文件覆盖新建数据库同名文件，接着，启动SQL Server Service Manager。对Master数据库将系统表设置为可更改状态   
Use Master   
Go   
sp\_configure 'allow updates', 1   
reconfigure with override   
Go   
将数据库设为紧急状态：   
update sysdatabases set status = 32768 where database '   
停止并重新启动SQL Server Service Manager，并重建Log文件：   
DBCC TRACEON (3604)   
DBCC REBUILD\_LOG(' test ','test \_log\_ldf')   
将数据库设置为单用户模式，然后进行检测：   
sp\_dboption ' test ', 'single user', 'true'   
DBCC CHECKDB(' test ')   
Go   
此数据库执行CHECKDB的过程中发现一些表的索引被破坏，于是针对具体的表进行重建索引的操作：   
DBCC DBREINDEX（表名）   
如执行以上操作仍然不能解决，若索引破坏的表是临时表或不是关键表，则可从新建账套中引入，若是主表，则可能通过近期的备份来（部份）恢复。若没有一个备份，则无法修复。   
  
1.3 SQL Server数据库为什么易损坏呢？   
以下是微软提供的一些可能引起数据库损坏的原因及一些预防措施：   
操作问题，包括冷起动机器、热拔硬盘、删除一些数据库文件；   
硬件问题，包括磁盘控制器的问题；   
操作系统问题，包括与系统相关的一些致命错误。   
  
1.4 预防措施：   
1、定期/不定期执行CHKDSK(不带参数)，以检测硬盘物理结构并修复一些CHKDSK报告的问题；   
2、常备份数据。   
  
1.5 应用数据库修复举例   
declare @databasename varchar(255)   
set @databasename='AIS20021224170730'------一定要手工输入   
---------执行一般性修复还存在问题时,进行允许数据丢失的修复   
---------许数据丢失的修复要求在单用户下进行,此时请退出中间层,客户端,sql的其他模块   
---所有功能退出,在查询分析器master里设置数据库为单用户   
  
exec sp\_dboption @databasename, N'single', N'true'   
  
-----在查询分析器master里,进行修复数据库   
dbcc checkdb(@databasename,REPAIR\_ALLOW\_DATA\_LOSS)   
dbcc checkdb(@databasename,REPAIR\_REBUILD)   
------还原数据库状态   
exec sp\_dboption @databasename, N'single', N'false'   
  
  
第2章数据库日志损坏的修复   
请遵照如下步骤来试图重建数据库事务日志.   
  
注意: 由于事务日志丢失, 数据库可能有没有提交的数据.   
  
注:都要替换成真实的数据库名字   
  
2.1 步骤1:   
  
创建一个新的数据库,命名为原来数据库的名字.   
  
2.2步骤2:   
  
停止SQL Server   
  
2.3步骤3:   
  
把老数据库的MDF文件替换新数据库的相应的MDF文件, 并把LDF文件删除   
  
2.4步骤4:   
  
重新启动SQL Server 服务,然后运行如下命令:   
  
Use Master   
  
Go   
  
sp\_configure 'allow updates', 1   
  
reconfigure with override   
  
Go   
  
begin tran   
  
update sysdatabases set status = 32768 where db\_name'   
  
-- Verify one row is updated before committing   
  
commit tran   
  
2.5步骤5:   
  
停止SQL然后重新启动SQL Server 服务,然后运行如下命令:   
  
DBCC TRACEON (3604)   
  
DBCC REBUILD\_LOG('db\_name','c:\mssql7\data\dbxxx\_3.LDF')   
  
Go   
  
2.6步骤6:   
  
停止SQL然后重新启动SQL Server 服务,然后运行:   
  
use master   
  
update sysdatabases set status = 8 where   
Go   
  
sp\_configure 'allow updates', 0   
  
reconfigure with override   
  
Go   
  
2.7步骤7:   
运行dbcc checkdb(db\_name)检查数据库的完整性.   
  
第3章 数据库质疑的一般处理   
1、执行如下SQL（打开修改系统表的开关）：   
EXEC sp\_configure 'allow updates', 1   
RECONFIGURE WITH OVERRIDE   
2、修改数据库Master中的表：sysdatabases   
将 status字段数值更改为4   
3、再执行如下SQL：   
EXEC sp\_configure 'allow updates', 0   
RECONFIGURE WITH OVERRIDE。