数据库项目文档

目录

1.SQL 常用对象整理

好的博客学习

http://blog.csdn.net/zjcxc
http://blog.csdn.net/fredrickhu
http://blog.csdn.net/happyflystone
http://blog.csdn.net/htl258
http://blog.csdn.net/wufeng4552
http://blog.csdn.net/roy_88

数据库项目文档

http://msdn.microsoft.com/zh-cn/library/windowsazure/ff951625 **系统表详解**

系统表的整理

1.1.1 Sysobjects 在每个数据库中都存在此表。存放着数据库中的所有对象，如表，列，索引等等。字段 xtype 代表对象类型，u 代表用户表，p 代表存储过程，tr 代表触发器。name 字段表示对象名称。 如：`select * from sysobjects where xtype='u'`

1.1.2 Syscolumns 在每个数据库中都存在此表。name:名称。id:该列所属的表对象 ID，或与该参数关联的存储过程 ID. 如：`select col.name,obj.name from syscolumns col join sysobjects obj on col.id = obj.id where obj.xtype = 'u' and obj.name like 'Dim_%' order by col.name`

1.1.3 Systypes 对于每种系统提供数据类型和用户定义数据类型，均包含一行信息。该表存储在每个数据库中。这些是系统提供的数据类型及其 ID 号。

1.1.4 Spt_values 相当于一个数字辅助表，在 sql 中主要用到 number 这个字段。列名分别为名称、值、类型、下限、上限、状态类型。列的取值含义：D=Database Option P=Projection DBR=Database Role DC=Database Replication I=Index L=Locks V=Device Type 因为比较多，无法一一列举。其中类型 P 较为特殊，它只是 0-2047(与版本有关)之间的数字的简单列表，作为对所有类型之间关系的预测。巧用 master..spt_values 表输出数字或者时间常量表 2009-10-16 14:32 。sql 开发中经常需要使用数字或者时间的常量表。比如，输出一年的月份表，输出 1000 以内的自然数等等。数量连续且不超过 2048。那么使用 master..spt_values 表就会再也方便不过了。

1.1.5 Syscoments：包含数据库中每个视图、规则、默认值、触发器、CHECK 约束、DEFAULT 约束和存储过程的项。**text** 列包含原始的 SQL 定义语句。 `select A.* ,B.* from syscomments A join sysobjects B on A.id=B.id where B.xtype= 'TR''`

1.1.6 Sysdatabases:存放数据库的信息

1.1.7 Sysforeignkeys: 包含关于表定义中的 FOREIGN KEY 约束的信息。该表存储在每个数据库中

1.18 Object_ID返回架构范围内对象的数据库对象标识号。`OBJECT_ID(N'A',N'U') /*AF = 聚合函数(CLR) C = CHECK 约束 D = DEFAULT（约束或独立）F = FOREIGN KEY 约束FN = SQL 标量函数FS = 程序集(CLR) 标量函数FT = 程序集(CLR) 表值函数IF = SQL 内联表值函数IT = 内部表P = SQL 存储过程PC = 程序集(CLR) 存储过程PG = 计划指南PK = PRIMARY KEY 约束R = 规则（旧式，独立）RF = 复制筛选过程S = 系统基表SN = 同义词SQ = 服务队列TA = 程序集(CLR) DML 触发器TF = SQL 表值函数TR = SQL DML 触发器TT = 表类型U = 表（用户定义类型）UQ = UNIQUE 约束V = 视图X = 扩展存储过程`

如果是临时表的话，就这样处理

```
*/
if object_id('tempdb.dbo.#TempTable') is not null
drop table #TempTable
if object_id('tempdb..#TempTable') is not null
drop table #TempTable
```

获取列的基本信息:

```
SELECT
表名=case when a.colorder=1 then d.name else '' end,
表说明=case when a.colorder=1 then isnull(f.value,'') else '' end,
字段序号=a.colorder,
字段名=a.name,
标识=case when COLUMNPROPERTY( a.id,a.name,'IsIdentity')=1 then '√'else
'' end,
主键=case when exists(SELECT 1 FROM sysobjects where xtype='PK' and name
in (
SELECT name FROM sysindexes WHERE indid in(
SELECT indid FROM sysindexkeys WHERE id = a.id AND colid=a.colid
))) then '√' else '' end,
类型=b.name,
占用字节数=a.length,
长度=COLUMNPROPERTY(a.id,a.name,'PRECISION'),
小数位数=isnull(COLUMNPROPERTY(a.id,a.name,'Scale'),0),
允许空=case when a.isnullable=1 then '√'else '' end,
默认值=isnull(e.text,''),
字段说明=isnull(g.[value],'')
FROM syscolumns a
left join systypes b on a.xusertype=b.xusertype
inner join sysobjects d on a.id=d.id and d.xtype='U' and
d.name<>'dtproperties'
left join syscomments e on a.cdefault=e.id
left join sys.extended_properties g on a.id=g.major_id and
a.colid=g.minor_id
left join sys.extended_properties f on d.id=f.major_id and f.minor_id=0
where d.name='D' --如果只查询指定表,加上此条件
order by a.id,a.colorder

/*
```

```
    功能说明：  获取表的详细信息
    修改说明：   Create by LY on 2012-5-30
*/

SELECT TableName=CASE
                  WHEN C.column_id = 1 THEN O.name
                  ELSE N''
                END,
       TableDesc=Isnull(CASE
                          WHEN C.column_id = 1 THEN PTB.[value]
                        END, N''),
       Column_id=C.column_id,
       ColumnName=C.name,
       PrimaryKey=Isnull(IDX.PrimaryKey, N''),
       [IDENTITY]=CASE
                    WHEN C.is_identity = 1 THEN N'√'
                    ELSE N''
                  END,
       Computed=CASE
                  WHEN C.is_computed = 1 THEN N'√'
                  ELSE N''
                END,
       Type=T.name,
       Length=C.max_length,
       PRECISION=C.PRECISION,
       Scale=C.scale,
       NullAble=CASE
                  WHEN C.is_nullable = 1 THEN N'√'
                  ELSE N''
                END,
       [Default]=Isnull(D.definition, N''),
       ColumnDesc=Isnull(PFD.[value], N''),
       IndexName=Isnull(IDX.IndexName, N''),
       IndexSort=Isnull(IDX.Sort, N''),
       Create_Date=O.Create_Date,
       Modify_Date=O.Modify_date
FROM   sys.columns C
       INNER JOIN sys.objects O
         ON C.[object_id] = O.[object_id]
            AND O.type = 'U'
            AND O.is_ms_shipped = 0
```

```
        INNER JOIN sys.types T
          ON C.user_type_id = T.user_type_id
        LEFT JOIN sys.default_constraints D
          ON C.[object_id] = D.parent_object_id
             AND C.column_id = D.parent_column_id
             AND C.default_object_id = D.[object_id]
        LEFT JOIN sys.extended_properties PFD
          ON PFD.class = 1
             AND C.[object_id] = PFD.major_id
             AND C.column_id = PFD.minor_id
        -- AND PFD.name='Caption' -- 字段说明对应的描述名称(一个字段可以添加多个不同name的描述)
        LEFT JOIN sys.extended_properties PTB
          ON PTB.class = 1
             AND PTB.minor_id = 0
             AND C.[object_id] = PTB.major_id
        -- AND PFD.name='Caption' -- 表说明对应的描述名称(一个表可以添加多个不同name的描述) bitsCN.Com网管联盟
        LEFT JOIN -- 索引及主键信息
        (SELECT IDXC.[object_id],
                IDXC.column_id,
                Sort=CASE Indexkey_property(IDXC.[object_id],
IDXC.index_id, IDXC.index_column_id, 'IsDescending')
                     WHEN 1 THEN 'DESC'
                     WHEN 0 THEN 'ASC'
                     ELSE ''
                    END,
                PrimaryKey=CASE
                           WHEN IDX.is_primary_key = 1 THEN N'√'
                           ELSE N''
                          END,
                IndexName=IDX.Name
         FROM   sys.indexes IDX
                INNER JOIN sys.index_columns IDXC
                  ON IDX.[object_id] = IDXC.[object_id]
                     AND IDX.index_id = IDXC.index_id
                LEFT JOIN sys.key_constraints KC
                  ON IDX.[object_id] = KC.[parent_object_id]
                     AND IDX.index_id = KC.unique_index_id
                INNER JOIN -- 对于一个列包含多个索引的情况,只显示第个索引信息
                (SELECT [object_id],
```

```
                    Column_id,
                    index_id=Min(index_id)
             FROM   sys.index_columns
             GROUP  BY [object_id],
                       Column_id) IDXCUQ
              ON IDXC.[object_id] = IDXCUQ.[object_id]
                 AND IDXC.Column_id = IDXCUQ.Column_id
                 AND IDXC.index_id = IDXCUQ.index_id) IDX
       ON C.[object_id] = IDX.[object_id]
          AND C.column_id = IDX.column_id
WHERE  O.name = N'LSXHDZP' -- 如果只查询指定表,加上此条件
ORDER  BY O.name,
          C.column_id

--查询用户表对象信息
select  Tab.Name as [表名],
       Tab.create_date as [创建时间],
       Tab.modify_date  as [最后修改时间],
       Col.Name  as [列名] ,
       Type.name as [数据类型],
       Col.max_length as [字段长度],
       CASE WHEN pk.is_primary_key= 1 THEN 'Y' ELSE 'N' end as [是否主
键],
       CASE WHEN Col.is_identity = 1 THEN 'Y' else 'N'end as [是否自增] ,
       identity_columns.seed_value as [自增种子],
       identity_columns.increment_value as [自增步长],
       case when Col.is_nullable = 1 then 'Y' else 'N' END AS [是否允许
为NULL],
       Def.text as [默认值],
       case when Col.is_computed = 1 then 'Y' else 'N' END as [是否计算
列] ,
       computed_columns.definition as [计算公式],Col_Desc.Value as [列备
注]
from
sys.objects Tab inner join sys.columns Col on Tab.object_id
=Col.object_id
             inner join sys.types Type on Col.system_type_id =
Type.system_type_id
             left join sys.identity_columns identity_columns on
Tab.object_id = identity_columns.object_id and Col.column_id =
identity_columns.column_id
```

```
            left join syscomments Def  on Col.default_object_id =
Def.ID
left join(
    select
index_columns.object_id,index_columns.column_id,indexes.is_primary_ke
y
    from sys.indexes  indexes inner join sys.index_columns index_columns
    on indexes.object_id = index_columns.object_id and indexes.index_id
= index_columns.index_id
    where indexes.is_primary_key = 1/*主键*/
) PK on Tab.object_id = PK.object_id AND Col.column_id = PK.column_id
left join sys.computed_columns  computed_columns on Tab.object_id
=computed_columns.object_id and Col.column_id =
computed_columns.column_id
left join sys.extended_properties Col_Desc on
Col_Desc.major_id=Tab.object_id and Col_Desc.minor_id=Col.Column_id
and Col_Desc.class=1
where Tab.type = 'U' and Tab.Name not like'sys%' and Type.name not like
'%sys%'
order by Tab.create_date



--查询所有视图
select views.Name as [视图名],Col.Name as [列名] ,Type.name as [数据类
型],Col.max_length as [字段长度]
       --,Col_Desc.Value as Col_Description
from sys.views views
inner join sys.columns Col on views.object_id  = Col.object_id
inner join sys.types Type on Col.system_type_id = Type.system_type_id
--left join sys.extended_properties Col_Desc
--    on Col_Desc.major_id=views.object_id and
Col_Desc.minor_id=Col.Column_id and Col_Desc.class=1
order by Create_Date



--查询外键约束
select FK_Name as [外键名],Parent_Tab_Name as [外键表],
    [外键列]=stuff((select ', '+[Parent_Col_Name] from (
       select    FK.name as FK_Name,Parent_Tab.Name as
Parent_Tab_Name,Parent_Col.Name as Parent_Col_Name,
```

```
            Referenced_Tab.Name as
Referenced_Tab_Name,Referenced_Col.Name as Referenced_Col_Name
       from sys.foreign_keys FK
       inner join sys.foreign_key_columns Col on FK.Object_ID =
Col.constraint_object_id
       inner join sys.objects Parent_Tab ON  Col.parent_object_id =
Parent_Tab.Object_ID and Parent_Tab.TYPE = 'U'
       inner join sys.columns Parent_Col on Parent_Tab.Object_ID =
Parent_Col.object_id
                                    and  Col.parent_column_id =
Parent_Col.column_id
       inner join sys.objects Referenced_Tab ON
Col.referenced_object_id = Referenced_Tab.Object_ID and
Referenced_Tab.TYPE = 'U'
       inner join sys.columns Referenced_Col on Referenced_Tab.Object_ID
= Referenced_Col.object_id
                                    and  Col.referenced_column_id =
Referenced_Col.column_id
    )t where FK_Name=tb.FK_Name  and Parent_Tab_Name = tb.Parent_Tab_Name
and Referenced_Tab_Name = tb.Referenced_Tab_Name   for xml path('')), 1,
1, ''),
    Referenced_Tab_Name as  [主键表],
    [主键列]=stuff((select ', '+[Referenced_Col_Name] from (
       select    FK.name as FK_Name,Parent_Tab.Name as
Parent_Tab_Name,Parent_Col.Name as Parent_Col_Name,
            Referenced_Tab.Name as
Referenced_Tab_Name,Referenced_Col.Name as Referenced_Col_Name
       from sys.foreign_keys FK
       inner join sys.foreign_key_columns Col on FK.Object_ID =
Col.constraint_object_id
       inner join sys.objects Parent_Tab ON  Col.parent_object_id =
Parent_Tab.Object_ID and Parent_Tab.TYPE = 'U'
       inner join sys.columns Parent_Col on Parent_Tab.Object_ID =
Parent_Col.object_id
                                    and  Col.parent_column_id =
Parent_Col.column_id
       inner join sys.objects Referenced_Tab ON
Col.referenced_object_id = Referenced_Tab.Object_ID and
Referenced_Tab.TYPE = 'U'
       inner join sys.columns Referenced_Col on Referenced_Tab.Object_ID
= Referenced_Col.object_id
```

```
                                        and  Col.referenced_column_id =
Referenced_Col.column_id
    )t where FK_Name=tb.FK_Name  and Parent_Tab_Name = tb.Parent_Tab_Name
and Referenced_Tab_Name = tb.Referenced_Tab_Name   for xml path('')), 1,
1, '')
    --as [外键列]
from (
    select    FK.name as FK_Name,Parent_Tab.Name as
Parent_Tab_Name,Parent_Col.Name as Parent_Col_Name,
          Referenced_Tab.Name as
Referenced_Tab_Name,Referenced_Col.Name as Referenced_Col_Name
    from sys.foreign_keys FK
    inner join sys.foreign_key_columns Col on FK.Object_ID =
Col.constraint_object_id
    inner join sys.objects Parent_Tab ON  Col.parent_object_id =
Parent_Tab.Object_ID and Parent_Tab.TYPE = 'U'
    inner join sys.columns Parent_Col on Parent_Tab.Object_ID =
Parent_Col.object_id
                                    and  Col.parent_column_id =
Parent_Col.column_id
    inner join sys.objects Referenced_Tab ON  Col.referenced_object_id =
Referenced_Tab.Object_ID and Referenced_Tab.TYPE = 'U'
    inner join sys.columns Referenced_Col on Referenced_Tab.Object_ID =
Referenced_Col.object_id
                                    and  Col.referenced_column_id =
Referenced_Col.column_id
)tb
group by FK_Name,Parent_Tab_Name,Referenced_Tab_Name


--查询所有存储过程
select Pr_Name  as [存储过程], [参数]=stuff((select ', '+[Parameter]
    from (
        select Pr.Name as Pr_Name,parameter.name +' ' +Type.Name + '
('+convert(varchar(32),parameter.max_length)+')' as Parameter
        from sys.procedures Pr left join
        sys.parameters parameter  on Pr.object_id = parameter.object_id
        inner join sys.types Type on parameter.system_type_id =
Type.system_type_id
        where type = 'P'
    ) t where Pr_Name=tb.Pr_Name for xml path('')), 1, 1, '')
```

```
from (
    select Pr.Name as Pr_Name,parameter.name +' ' +Type.Name + '
('+convert(varchar(32),parameter.max_length)+')' as Parameter
    from sys.procedures Pr left join
    sys.parameters parameter  on Pr.object_id = parameter.object_id
    inner join sys.types Type on parameter.system_type_id =
Type.system_type_id
    where type = 'P'
)tb
where Pr_Name not like 'sp_%' --and Pr_Name not like 'dt%'
group by Pr_Name
order by Pr_Name

--查询所有触发器
select triggers.name as [触发器],tables.name as [表
名],triggers.is_disabled as [是否禁用],
triggers.is_instead_of_trigger AS [触发器类型],
case when triggers.is_instead_of_trigger = 1 then 'INSTEAD OF'
     when triggers.is_instead_of_trigger = 0 then 'AFTER'
     else null
end as [触发器类型描述]
from sys.triggers triggers
inner join sys.tables tables on triggers.parent_id = tables.object_id
where triggers.type ='TR'
order by triggers.create_date

--查询所有索引
select    indexs.Tab_Name  as [表名],indexs.Index_Name as [索引
名] ,indexs.[Co_Names] as [索引列],
       Ind_Attribute.is_primary_key as [是否主
键],Ind_Attribute.is_unique AS [是否唯一键],
       Ind_Attribute.is_disabled AS [是否禁用]
 from (
    select Tab_Name,Index_Name, [Co_Names]=stuff((select ', '+[Co_Name]
from
    (    select tab.Name as Tab_Name,ind.Name as Index_Name,Col.Name as
Co_Name from sys.indexes ind
       inner join sys.tables tab on ind.Object_id = tab.object_id and
ind.type in (1,2)/*索引的类型：=堆/1=聚集/2=非聚集/3=XML*/
       inner join sys.index_columns index_columns on tab.object_id =
index_columns.object_id and ind.index_id = index_columns.index_id
```

```
        inner join sys.columns Col on tab.object_id = Col.object_id and
index_columns.column_id = Col.column_id
    ) t where Tab_Name=tb.Tab_Name and Index_Name=tb.Index_Name  for xml
path(''), 1, 1, '')
    from (
        select tab.Name as Tab_Name,ind.Name as Index_Name,Col.Name as
Co_Name from sys.indexes ind
        inner join sys.tables tab on ind.Object_id = tab.object_id and
ind.type in (1,2)/*索引的类型：=堆/1=聚集/2=非聚集/3=XML*/
        inner join sys.index_columns index_columns on tab.object_id =
index_columns.object_id and ind.index_id = index_columns.index_id
        inner join sys.columns Col on tab.object_id = Col.object_id and
index_columns.column_id = Col.column_id
    )tb
    where Tab_Name not like 'sys%'
    group by Tab_Name,Index_Name
) indexs inner join sys.indexes  Ind_Attribute on indexs.Index_Name =
Ind_Attribute.name
order by indexs.Tab_Name
```

数据表列的整理

```
use City;
go
SET NOCOUNT ON  /*使返回的结果中不包含有关受Transact-SQL 语句影响的行数的信息
*/
if OBJECT_ID(N'A',N'U') is not null drop table A
go
create table A
(
  ID int identity(1,1) primary key not null,
  Name nvarchar(100)
)
go
/*添加列*/
alter table A  add  Number nvarchar(100)
/*修改列的类型*/
alter table A  alter column Number int
/*删除列*/
```

```
alter table A  drop  column Number
select * from A
/*修改列的名称必须知道表名*/
exec sp_rename @objname='A.Name',@newname='NewName',@objtype='column';
go
select * from A
drop table A

Alter table dbo.A add constraint FK_A_Number
foreign key (Number) references dbo.Dim_Shop(ShopID)
/*添加列时不指定关键字[column],删除和修改需要指定*/
/*修改列名称时，执行的是一个存储过程，第一个参数是原始对象名，第二个是新的对象名，
第三个参数是修改对象的类型，有：column、database、index等*/
```

Case When 基本语法

```
case when then
格式：
case XX
  when YY then WW
  when YY then WW
  when YY then WW
  else//其他情况
  MM
end

如果没有else和及其子句，则返回NULL

--简单Case函数
CASE sex
        WHEN '1' THEN '男'
        WHEN '2' THEN '女'
ELSE '其他' END
解析：当sex和when中的条件{相等}时就返回，when后then的值

--Case搜索函数
CASE WHEN sex = '1' THEN '男'
        WHEN sex = '2' THEN '女'
ELSE '其他' END
```

解析：当when的表达式为{真}时，就返回then后的值

意思就是说，在要用sql进行辑判断的时候就想到用这个case when then else end 语句

1、而在执行的时候，就相当于遍历期间的每一行，然后进行IF else判断，
2、select:Case返回的结果在select中就相当于一列 ，
（一）、用【一条SQL语句】进行不同条件的分组，并将结果按分组显示(如工资的不同等级分组，并且按不同分组显示),比如数据库有字段【国家】【人口】，按照亚洲、非洲等进行分组、再比如，数据表字段有【薪水】，要按照不同等级的薪水进行分组等。
（二）、按照另外一种条件(该条件在数据库中没有这个字段)进行【分组】，并将结果按分组显示，
总结是：将结果按分组显示就要将条件和分组中都要写case when then语句

3、update:在进行不同条件进行不同处理的时候，如根据不同条件进行不同程度的更新，通常我们可以将这个更新语句写成多个，但是多个会出问题，因为，第一个更新语句对第二个更新语句是有冲突的，于是我们就用一条语句来实现
（一）、比如将字段值为1的换成2,将字段值为2的换成1

Covert Case 区别

CONVERT是专对SQL Server使用的，使日期与时间值，小数之间转换具有更宽的灵活性。
CAST是两种功能中更具ANSI标准的功能，即虽然更具便携性（比如，使用CAST的函数能更容易的被其它数据库软件使用），但功能相对弱一些。

Exec 与 Exec()及 Exec sp_executesql 的使用

Exec @sql 不符合语法规则，建议不使用
Exec (@sql) 符合语法规则，建议经常使用.

```
EXEC SP_EXECUTESQL
      @SQL,--相当于存储过程的函数体
      @PARAME_VALUE,--相当于存储过程中的参数定义
      @RETURNVALUE=@RETURN_VALUE OUTPUT—相当于在外面调用存储过程
```

MSSQL为我们提供了两种动态执行SQL语句的命令，分别是EXEC和sp_executesql;通常,sp_executesql则更具有优势，它提供了输入输出接口，而EXEC没有。还有一个最大的好处就是利用sp_executesql,能够重用执行计划,这就大大提供了执行性能(对于这个我在后面的例子中会详加说明),还可以编写更安全的代码。EXEC在某些情况下会更灵活。除非您有令人信服的理由使用EXEC，否侧尽量使用sp_executesql.

SqlServer 执行顺序

Group by HAVING where order By from select的执行顺序
from->where->Group by->HAVING->order By->select
记住：select 中的列别名只有order可以用，where都不能用

(8)SELECT (9) DISTINCT (11)<TOP_specification> <select_list>
(1)FROM <left_table>
(3) <join_type> JOIN <right_table>
(2) ON <join_condition>
(4)WHERE <where_condition>
(5)GROUP BY <group_by_list>
(6)WITH {CUBE | ROLLUP}
(7)HAVING <having_condition>
(10)ORDER BY <order_by_list>

每个步骤产生一个虚拟表，该虚拟表被用作下一个步骤的输入。只有最后一步生成的表返回给调用者。如果没有某一子句，则跳过相应的步骤。

1. FROM:对FROM子句中的前两个表执行笛卡尔积，生成虚拟表VT1。
2. ON:对VT1应用ON筛选器。只有那些使<join_condition>为真的行才被插入VT2。
3. OUTER(JOIN):如果指定了OUTER JOIN，保留表中未找到匹配的行将作为外部行添加到VT2，生成VT3。如果FROM子句包含两个以上的表，则对上一个联接生成的结果表和下一个表重复执行步骤1到步骤3，直到处理完所有的表为止。
4. 对VT3应用WHERE筛选器。只有使<where_condition>为TRUE的行才被插入VT4。
5. GROUP BY:按GROUP BY 子句中的列列表对VT4中的行分组，生成VT5。
6. CUBE|ROLLUP:把超组插入VT5，生成VT6。
7. HAVING:对VT6应用HAVING筛选器。只有使<having_condition>为TRUE的组才会被插入VT7。
8. SELECT:处理SELECT列表，产生VT8。
9. DISTINCT:将重复的行从VT8中移除，产生VT9。
10. ORDER BY:将VT9中的行按ORDER BY子句中的列列表排序，生成一个有表(VC10)。
11. TOP:从VC10的开始处选择指定数量或比例的行，生成表VT11,并返回给调用者。

注释：

1、from->on->join->where->group by->
cube|rullup->having->select->distinct->order by ->top

1、 join即为笛卡尔积，而LEFT、RIGHT、FULL中的一种OUTER JOIN
2、 where不能使用聚合函数，因为还没执行分组，此外也不能用select列表的别名
3、 可以说，group by 后面的结果才是最后的结果，即，select 列表中的列必须在group by后面出现，因为先执行的是group by
4、 Having选择符合条件的组

5、 只有order by可以访问select列表的别名

6、 Order by返回的是一个游标，不是返回表，使用了ORDER BY子句的查询不能用作表表达式。表表达式包括：视图、内联表值函数、子查询、派生表和共用表表达式(CTE)。但是可以通过访问游标最前面指定的函数集用top N percent,这时才返回表。

Order by A,B:

order by A，B表示：首先整个结果集按照A进行升序排序，然后对于A值相同的结果集再进行B升序排序

表添加默认值

```
create table student(
    stuId int identity(1,1) primary key,--想在这里加入主键，这个我已经写好了！
    stuName varchar(10) unique not null,--想在这里加入唯一约束，怎么写？
    stuAge int Check(stuAge between 18 and 100),--想在这里加入检查约束怎么写？大于=18，小于=100就好。
    stuAddress varchar(20) default 'china',--想在这里加入默认约束怎么？
    stuDel varchar(20)
)
go
```

Insert into select 语句与 select into from 语句

语句形式为：Insert into Table2(field1,field2,...) select value1,value2,... from Table1

要求目标表Table2必须存在，由于目标表Table2已经存在，所以我们除了插入源表Table1的字段外，还可以插入常量。

语句形式为：SELECT vale1, value2 into Table2 from Table1

要求目标表Table2不存在，因为在插入时会自动创建表Table2，并将Table1中指定字段数据复制到Table2中。

Set 和 Select 赋值区别

1.假如你是标准SQL的开发者，那么请使用SET吧，因为SET是ANSI标准的SQL语句，SELECT

不是。
2.你可以使用SELECT一次给两个以前变量赋值，但是SET不能。
3.你可能会意识到系统变量@@ERROR and @@ROWCOUNT必须要在一句SQL语句中捕获。并且是在DML语句（Select、INSERT, UPDATE, DELETE等）之后立即捕获
4.SET和SELECT还有一个区别是，当使用查询出来的值赋值给变量时，SET和SELECT都可以实现，但当查询出的值为多个是，SET会提示错误，但SELECT不会，只会接受最后一个值。这点很重要，也是很多程序Bug容易被忽略的地方。

完整的对象名称由四个标识符组成

服务器名称、数据库名称、架构名称和对象名称。其格式如下：
[[[server.] [database] .] [schema_name] .] object_name
服务器、数据库和所有者的名称即所谓的对象名称限定符。引用对象时，不必指定服务器、数据库和所有者（省略时的默认值分别为：当前连接服务器、当前连接数据库、当前连接用户默认架构）。可以用句点标记它们的位置来省略限定符。对象名称的有效格式包括以下几种：

```
server.database.schema_name.object_name
server.database..object_name
server..schema_name.object_name
server...object_name
database.schema_name.object_name
database..object_name
schema_name.object_name
object_name
```

一个完整的对象名称唯一地决定一个SQLServer数据库对象。

字符串操作:

1、 查找某个字符的位置[IndexOf()]： `charindex('a','sas') 2`
2、 ASCII,获取一个字符的 ASCII 值
3、 Stuff[Insert]:将一个字符串中的字符插入的另一个字符串中
Stuff('12345',2,3,'abcd'),将第一个字符串从第二个位置开始，删除 3 个字符，然后将第二个字符串从第 2 个位置插入：1abcd5.
4、 Left('123456',4),获取字符串左边的 4 个字符。1234
5、 Right（）和 left 相反
6、 'Replicate('0',4)，对字符 0 重复 4 次=>0000
7、 Len（'fdsf '）长度：4

8、 Reverse（‘12345 ‘）反转=>54321

9、 Lower 和 upper 大小写

10、 Rtrim(‘fdfd ’)和 Ltrim(‘ fdfd’)，去掉左边的空格和去掉右边的空格=>fdfd

11、 QuotName(),相当于 C#中的@符号，使得字符串中的一些特殊字符有效

12、 patindex 可以模式匹配，即使用通配符，charindex 只能字符匹配。

一，将字符串 BEIJING 转为：

B

E

I

J

I

N

G

用 substring 分割字符

[1]、substring(1,2,3) 1:为你要分割的字符串,2:起始位置，3：长度

注释：第一个字符的位置为 1，不是为 0，这和 C#中是不一样的。

[2]、select 后跟引号的时候，引号相当于(),也都是成双出现的,""代表一个',""""代表两个'。

注释：两个''中间是字符串，里面没东西表示 null

二、计算某字符串中某字符的个数

首先计算该字符串的长度，然后将该字符串删去那个字符(替换成空值)，然后再次计算字符串，两次计算的结果相减即可。

Replace 替换字符，也可以说是删除字符

[3]、replace(1,2,3) 1:带处理的字符串，2:要替换的字符,3:要将 2 字符替换成什么字符

例如：取最后一个||的内容。

```
select right('上海||中国||人民测试中||afdd||TTTTT',charindex('||',reverse('上海||中国||人民测试中||afdd||TTTTT'))-1)

SELECT ID,
      RIGHT(type,CHARINDEX('|',REVERSE(type))-1) AS type
FROM tb
```

1:replace 函数

第一个参数你的字符串，第二个参数你想替换的部分，第三个参数你要替换成什么

```
select replace('lihan','a','b')

------------------------------
lihbn

（所影响的行数为 1 行）
```

==========

2:substring函数(SQL Server 2005中)

第一个参数你的字符串，第二个是开始替换位置，第三个是你要获取的字符长度！第一个字符位置为1，不是0

```
select substring('lihan',1,3);
-----
lih

（所影响的行数为 1 行）
```

==========

3:charindex函数

第一个参数你要查找的char,第二个参数你被查找的字符串,第三个参数是指从哪个位置开始查找， 返回参数一在参数二的位置

```
select  charindex('a','lihan',1)
-----------
4

（所影响的行数为 1 行）
```

==========

4：ASCII函数

返回字符表达式中最左侧的字符的 ASCII 代码值。

```
select ASCII('lihan')
-----------
108

（所影响的行数为 1 行）
```

==========

5：nchar函数

根据 Unicode 标准的定义，返回具有指定的整数代码的 Unicode 字符。

参数是介于 0 与 65535 之间的正整数。如果指定了超出此范围的值，将返回 NULL。

```
select nchar(3213)
----
unicode字符

（所影响的行数为 1 行）
```

==========

6：soundex

返回一个由四个字符组成的代码 (SOUNDEX)，用于评估两个字符串的相似性。

```
SELECT SOUNDEX ('lihan'), SOUNDEX ('lihon');
----- -----
L546  L542

（所影响的行数为 1 行）
```

==========

7：char

参数为介于 0 和 255 之间的整数。如果该整数表达式不在此范围内，将返回 NULL 值。

```
SELECT char(125)
----
}

（所影响的行数为 1 行）
```

==========

8：str函数

第一个参数必须为数字，第二个参数表示转化成char型占的位置，小于参数一位置返回*，大于右对齐

```
SELECT str(12345,3)
----
***

（所影响的行数为 1 行）

SELECT str(12345,12)
------------
       12345

（所影响的行数为 1 行）
```

==========

9：difference函数

返回一个整数值，指示两个字符表达式的 SOUNDEX 值之间的差异。

返回的整数是 SOUNDEX 值中相同字符的个数。返回的值从 0 到 4 不等：0 表示几乎不同或完全不同，4 表示几乎相同或完全相同。

```
SELECT difference('lihan','liha')
-----------
3

（所影响的行数为 1 行）
```

```
==============================================================
10：stuff函数（四个参数）
函数将字符串插入另一字符串。它在第一个字符串中从开始位置删除指定长度的字符；然后将第二个字符串插入第一个字符串的开始位置。
SELECT stuff('lihan',2,3,'lihan')
--------
llihann

（所影响的行数为 1 行）
==============================================================

11：left函数
返回最左边N个字符，由参数决定
select left('lihan',4)
-----
liha

（所影响的行数为 1 行）
==============================================================

12 right函数
返回最右边N个字符，由参数决定
select right('lihan',4)
-----
ihan

（所影响的行数为 1 行）
==============================================================

13：replicate函数
我的认为是把参数一复制参数二次
select replicate('lihan',4)
--------------------
lihanlihanlihanlihan

（所影响的行数为 1 行）

==============================================================

14：len函数
```

返回参数长度

```
select len('lihan')
-----------
5

（所影响的行数为 1 行）
```

==========

15：reverse函数

反转字符串

```
select reverse('lihan')
-----
nahil

（所影响的行数为 1 行）
```

==========

16：lower和upper函数

参数大小写转化

```
select lower(upper('lihan'))
--------------------
lihan

（所影响的行数为 1 行）
```

==========

17：ltrim和rtrim函数

删除左边空格和右面空格

```
select ltrim('    lihan              ')
---------------------------
lihan

（所影响的行数为 1 行）
select rtrim('    lihan')
---------
    lihan
```

（所影响的行数为 1 行）

表中列的扩展属性:

列的扩展属性：说明等
获取有两种方式：
一、sys.extended_properties.value　通过系统视图来实现
二、通过系统函数来获取

```
USE AdventureWorks;
GO
SELECT objtype, objname, name, value
FROM fn_listextendedproperty (NULL, 'schema', 'Production', 'table', 'ScrapReason', 'column',
default);
GO
```

修改、删除和更新扩展属性：
sp_updateextendedproperty、sp_dropextendedproperty、sp_addextendedproperty

是否存在[存储过程][表][函数]等:

存储过程

```
if(exists(select * from sysobjects where name='存储过程的名字' and Type='P'))
```

触发器：

```
   select * from sysobjects where id=object_id(N'触发器的名字') and
objectproperty(id,N'IsTrigger')=1
```

如果判断用户表格的话，用IsUserTable 代替 上面的IsTrigger

函数

```
select * from sysobjects where id = object_id(N'[dbo].[USER_Fun]') and (type = 'FN' or type =
'TF'))
--判断是否存在USER_Fun这个用户函数(注意此处的type 有两种,分别是'TF'-Table-valued
```

Function 表值函数 或'FN'-Scalar-valued Function 标量值函数)

函数和存储过程的区别:

1、无论系统函数和UDF,在系统启动是就进行编译并加载，所以UDF的效率比SP要高，SP只是在调用时才加载（扩展的存储过程除外）。
2、有些函数getdate,exec..都在UDF里不能用！所以函数中限制较多，存储过程限制少
3、函数必须有返回值，SP则不一定。
4、函数可以包括在FROM子句中，SP则不可以。
5、在SQL SERVE2000中SP可以DEBUG,UDF不可以。

SqlServer 数据库中一些基本的概念(常用)

一、变量要先声明：declare @InsertCount int
给变量赋值 Set @InsertCount=1
屏蔽用‘--’，如：
--fdsfdsfdsfdsfdf
一、数据类型转换：cast
'张三'+cast(15 as varchar(10))
二、循环
while 条件
begin
执行操作
end
1、break和continue是一样用的
2、如果条件是一个sql语句必须用()包起来

三、Exists ：相关查询，即子查询中要用到外层表的某个属性，【存在】
先获取外层查询的第一个元组，然后将这个元组的属性带入子查询中进行逻辑查询，如果为真即选择，不为真则转入外层查询的第二个元组。

四、将空值转换为指定的值
coalesce(sd,'梁勇') 【联合、合并】
五、select 列1 as one,列2 two

其中one和two只能用来在现实结果的时候列名为one和two而已，而不能替代相应列而在where条件语句中进行使用

六、将多表中的行纵向组合显示，当然要显示的列的属性每个表的定义都要一样，比如都是字符型的，顺序和个数也要一样。

```
select UserName from A
union all
select username from B
```

union all将显示结果中的原始值
union 就是想显示结果加一个distinct，显示的结果没有重复的值，建议用union all
UNION的结果集列名与第一个select语句中的结果集中的列名相同，其他select语句的结果集列名被忽略

如果要横向的话就用连接吧

七、等值链接，是先获取笛卡尔积，然后通过等值条件获取符合的行。
貌似是from 后出现两个表或多个表就是笛卡尔积，还是用内链接或者左右链接吧，速度最快

八、用一张的数据向另一张表中添加多条记录

```
insert into TableA(field1,field2)
select field1,field2
from TableB
where ID=10
```

九、用一张表的记录去更新另一张表的记录

```
update A
set A.Name=B.Name
set A.Sex=B.Sex
from A,B
where
A.ID=B.ID
```

十、行转列或列转行都是根据case来进行的

十一、公共表表达式CTE

优点：没有像临时表那样耗性能、也没有像子查询那样可读性差、增强了可维护性和效率。

with 别名1 as(selec语句),别名2 as(select语句)

一个查询分析器中可以定义多个CET，也可以嵌套，【With前面的语句要以分号结束，】

十二、count(*)获取的是行数，假设某一行全是NULL，那么count(*)能累加到这行，但是若是Count(列名)，则不累加这行

聚集函数会忽略NULL的

十三.

一、如果更新失败就插入，这样就免了先判断下是否存在，存在就更新，不存在就插入

```
        UPDATE Production.UnitMeasure
    SET Name = @Name
    WHERE UnitMeasureCode = @UnitMeasureCode
-- Insert the row if the UPDATE statement failed.
    IF (@@ROWCOUNT = 0 )
    BEGIN
        INSERT INTO Production.UnitMeasure (UnitMeasureCode, Name)
        VALUES (@UnitMeasureCode, @Name)
    END
```

二、

1、MERGE Production.UnitMeasure AS target

2、USING (SELECT @UnitMeasureCode, @Name) AS source (UnitMeasureCode, Name)

3、ON (target.UnitMeasureCode = source.UnitMeasureCode)

其中第2句的意思是SELECT @UnitMeasureCode, @Name表示获取者两个值，并作为一个结果集AS source (UnitMeasureCode, Name)表示，将上面的结果集另取一个名称，并且给这个新的结果集中的列重新命名为：UnitMeasureCode和 Name，这样后面在使用这个新结果集的时候就可以用这个新结果集新的字段名

常用的小细节

一、

判断存储过程是否存在：

```
if exists(select * from sysobjects where name='you Pro name' and type='you procedure type')
begin
end
```

Is Same to next

```
if Object_Id(N'you procedure's name',N'you procedure's type') is not null
begin
end
```

then,the way of judge the UserTable(Table Createed by youself) is Exists is next

---

if object_id(N'CityTable',N'U') is not null

二、Cultivate the habit of(养成XX习惯) Write Next

1、when Exec the oprater of select ,you can write the Statement(语句)：

SET NOCOUNT ON(这样就不会每次都要在服务器运行并计算本次操作影响的行数)

三、在写SQL语句时，在前面加个

USE DBName;(注意这里有个分号)

GO

四、每次进行操作后对后面的操作是有影响的，也就是只有前面的操作才有后面的操作时，记得要作为一个批处理进行提交。即。go

五、如果不能用Order by 就在前面加个top 100 percent即可

六、查询表，存储过程或函数等时加上模式名，如，dbo.ProName,一般在查询，删除，添加，引用等的时候，除了创建以外的使用情况都加吧。

七、带有 group by语句的 select 后面跟着的字段的聚集函数都是针对于某个分组进行的，而不是整个结果集。

八、distinct的用法，只能用一次，且在最前面，去除重复的行，而不是去除重复的列

简单的数据拆分

```
/*标题：简单数据拆分(version 2.0)
作者：爱新觉罗.毓华(十八年风雨,守得冰山雪莲花开)
时间：-05-07
地点：重庆航天职业学院
描述：

有表tb，如下：
id          value
----------- -----------
1           aa,bb
2           aaa,bbb,ccc
欲按id,分拆value列，分拆后结果如下：
```

```
id          value
----------- --------
1           aa
1           bb
2           aaa
2           bbb
2           ccc
*/

--1. 旧的解决方法(sql server 2000)

create table tb(id int,value varchar(30))
insert into tb values(1,'aa,bb')
insert into tb values(2,'aaa,bbb,ccc')
go

--方法.使用临时表完成
SELECT TOP 8000 id = IDENTITY(int, 1, 1) INTO # FROM syscolumns a,
syscolumns b

SELECT A.id, value = SUBSTRING(A.[value], B.id, CHARINDEX(',', A.[value]
+ ',', B.id) - B.id)
FROM tb A, # B
WHERE SUBSTRING(',' + A.[value], B.id, 1) = ','

DROP TABLE #

--方法.如果数据量小，可不使用临时表
select a.id , value = substring(a.value , b.number , charindex(',' ,
a.value + ',' , b.number) - b.number)
from tb a join master..spt_values  b
on b.type='p' and b.number between 1 and len(a.value)
where substring(',' + a.value , b.number , 1) = ','

--2. 新的解决方法(sql server 2005)
create table tb(id int,value varchar(30))
insert into tb values(1,'aa,bb')
insert into tb values(2,'aaa,bbb,ccc')
go

--方法.使用xml完成
```

```
SELECT A.id, B.value FROM
(
  SELECT id, [value] = CONVERT(xml,'<root><v>' + REPLACE([value], ',',
'</v><v>') + '</v></root>') FROM tb
) A OUTER APPLY
(
  SELECT value = N.v.value('.', 'varchar(100)') FROM
A.[value].nodes('/root/v') N(v)
) B

--方法.使用CTE完成
;with tt as
(select id,[value]=cast(left([value],charindex(',',[value]+',')-1) as
nvarchar(100)),Split=cast(stuff([value]+',',1,charindex(',',[value]+'
,'),'') as nvarchar(100)) from tb
union all
select id,[value]=cast(left(Split,charindex(',',Split)-1) as
nvarchar(100)),Split= cast(stuff(Split,1,charindex(',',Split),'') as
nvarchar(100)) from tt where split>''
)
select id,[value] from tt order by id option (MAXRECURSION 0)


DROP TABLE tb

/*
id          value
----------- ------------------------------
1           aa
1           bb
2           aaa
2           bbb
2           ccc

(5 行受影响)
*/
```

排序问题

```
create table tb(col varchar(10))
```

```
insert tb
select '1' union all
select '2' union all
select '3' union all
select '11' union all
select '12' union all
select '13' union all
select 'B1' union all
select 'B2' union all
select 'B3' union all
select 'B11' union all
select 'B12' union all
select 'B13' union all
select 'A1' union all
select 'A2' union all
select 'A3' union all
select 'A11' union all
select 'A12' union all
select 'A13'
```

存储过程实现递归

```
select * from tb
order by case when ISNUMERIC(col) =1 then '' else left(col,1) end,
cast(case when ISNUMERIC(col) =0 then stuff(col,1,1,'') else col end  as
int)

drop table dbo.Example


create table Example(ID int,ParentID int)

insert Example
select 0,null union
select 1,0 union
select 2,0 union
select 3,1 union
select 4,6 union
select 5,2
```

```
--取其某个ID为父节点的树结构的SQL如下：
create procedure display(@ID int) as
begin

WITH Example_Table(ID,ParentID,lv)
AS
(

        --取根节点放入临时表
        SELECT ID,ParentID,0 FROM Example WHERE ID=@ID

        --根据已取到的数据递归取其字节点的数据
        UNION ALL
        SELECT A.ID,A.ParentID,B.lv+1 FROM Example A INNER JOIN
Example_Table B ON A.ParentID = B.ID

)

SELECT * FROM Example_Table

end

drop table tb
go
```

SQL 另类排序问题

```
/*
如果不考虑orderindex，那么排序是从开始直接记录数为止。那么在考虑orderindex时，
借助系统表将从到这个表的记录数的所有计数罗列出来，将其中在表里orderindex这个
字段存在的去除，从小到大做个标记列。

将表中orderindex为null的按日期升序做标记，是为了和之前计数表统一。

然后将两个表按计数列合并，当然这些都是orderindex在原表为null的数据，
查询出结果集后再将原表orderindex不为null的union all，得到新的排序。
*/
create table tb(name  char(10),createtime datetime,orderindex int)
```

```
insert into tb
select 'aa',dateadd(hh,3,getdate()),1 union all
select 'cc',dateadd(hh,2,getdate()),null union all
select 'dd',dateadd(hh,1,getdate()),null union all
select 'ee',dateadd(hh,5,getdate()),null union all
select 'tt',dateadd(hh,4,getdate()),3
go

;with ac1 as
(
    select distinct number,rid=row_number() over (order by getdate())
    from master..spt_values
    where number between 1 and (select count(*) from tb)
       and number not in (select orderindex from tb where orderindex is
not null)
       and [type] = 'p'
)
--select * from ac1
,ac2 as
(
    select *,px=row_number() over (order by createtime)
    from tb
    where orderindex is null
)

select a.name,a.createtime,b.number
from ac2 a join ac1 b on a.px = b.rid
union all
select *
from tb
where orderindex is not null
```

SQL 面试题(Not Exists 与 Exists 的区别与算法)

```
/*
1 删除A中,同时存在于A,B中的数据.
2 用联合查询汇总A,B数据的结果集,要求查询结果剔除重复记录.
3 将B表中存在,而A表中不存在的数据添加到A中.
```

数据库项目文档

数据库的表如下：

```
table A:
  a
  1
  2

table B:
  b
  2
  3

*/  --存在与不存在的方法
--1.
delete from A where exists(select 1 from B where b=A.a)
--2.
select * from A
union
select * from B
--3.
insert into A select * from B where not exists(select 1 from A where a=B.b)
```

不规则字段，SQL 排序问题

```
create table tb23(col varchar(10))
insert tb23 select '1.1.10'
insert tb23 select '2.1.9'
insert tb23 select '1.1.8'
insert tb23 select '1.1.11'

select * from tb23
order by cast(Parsename(col,3) as int),
        cast(Parsename(col,2) as int),
        cast(Parsename(col,1) as int)


;with cte as(
select id,(case len(id)-len(replace(id,'.','')) when 0 then id+'.0.0'
```

```
when 1 then id+'.0' else id end) as o from tb
)
select id from cte order by
convert(int,parsename(o,3)),convert(int,parsename(o,2)),convert(int,p
arsename(o,1))


select * from tb
order by Parsename(col,3)+0,Parsename(col,2)+0,Parsename(col,1)+0
```

求合计，左边加个合计，右边也加个合计

```
--> 测试数据：#tb1
create table #tb1([ID] int,[pType] varchar(5))
insert #tb1
select 1,'类型A' union all
select 2,'类型B'
--> 测试数据：#tb2
create table #tb2([ID] int,[pTypeId] int,[proCount] int,[proDate]
datetime)
insert #tb2
select 1,1,10,'2011-7-1' union all
select 2,2,15,'2011-7-1' union all
select 3,1,12,'2011-7-2' union all
select 4,2,13,'2011-7-2'

DECLARE @s varchar(8000);
SET @s = '';

SELECT @s = @s + ',SUM(CASE WHEN proDate=''' + proDate+''' THEN proCount
ELSE 0 END) AS ['+proDate+']'
FROM (SELECT DISTINCT CONVERT(varchar(10),proDate,120) AS proDate FROM
#tb2) AS A;

SET @s = 'SELECT ISNULL(A.pType,''合计'') AS pType' + @s +',SUM(proCount)
AS proCount'
        +' FROM #tb1 AS A JOIN #tb2 AS B'
        +' ON A.ID=B.pTypeID'
```

```
        +' GROUP BY A.pType WITH ROLLUP';

print @s
--EXEC(@s);
SELECT  ISNULL(A.pType,'合计') AS pType,
       SUM(CASE WHEN proDate='2011-07-01' THEN proCount ELSE 0 END) AS
[2011-07-01],
       SUM(CASE WHEN proDate='2011-07-02' THEN proCount ELSE 0 END) AS
[2011-07-02],
       SUM(proCount) AS proCount FROM #tb1
 AS A JOIN #tb2 AS B ON A.ID=B.pTypeID GROUP BY A.pType WITH ROLLUP

DROP TABLE #tb1,#tb2;


create table A(id int ,pType varchar(8))
--  ID pType
insert a
select 1 ,'类型A' union all
select 2 ,'类型B'

create table B(id int,pTypeId int,proCount int,proDate datetime)
 -- ID pTypeId proCount proDate
insert b
select 1 ,1 ,10 ,'2011-7-1' union all
select 2 ,2 ,15 ,'2011-7-1' union all
select 3 ,1 ,12 ,'2011-7-2' union all
select 4 ,2 ,13 ,'2011-7-2'

declare @sql varchar(8000)
set @sql = 'select isnull(pType,''合计'')类型名称 '
select @sql = @sql + ' ,sum(case proDate when ''' + ltrim(proDate) + '''
then proCount else 0 end) [' + convert(varchar(10),proDate,23) + ']'
from (select distinct proDate from b) as ta
set @sql = @sql + ',sum(proCount)合计from (select
pTypeId,proCount,proDate,pType from b,a where b.pTypeId=a.id)tb group by
pType with rollup'
exec(@sql)


/*
```

```
类型名称    2011-07-01  2011-07-02  合计
-------- ----------- ----------- -----------
类型A      10          12          22
类型B      15          13          28
合计       25          25          50
*/


select isnull(课程,'合计'),sum(分数) from tb group by 课程
with ROLLUP
```

数据分类

```
SQL 分类：
DDL—数据定义语言(CREATE，ALTER，DROP，DECLARE)
DML—数据操纵语言(SELECT，DELETE，UPDATE，INSERT)
DCL—数据控制语言(GRANT，REVOKE，COMMIT，ROLLBACK)
```

Windows 自动定时执行任务的几种实现方法

Windows 自动定时执行任务，常见的方法有三种：

- Windows 任务计划程序
- Windows Service
- SQL Agent 的 Job

这三种方法大多数人都用过，我在这里只做一个比较简单的介绍和小结，后续，我会用一个轻量型的.Net Windows Service Jobs 的作为例子介绍如何定制 Windows Service 以及如何让 Windows Service 自动定时执行任务。

首先，我们来看：使用 Windows 任务计划程序 如何创建自动定时执行任务：

数据库项目文档

界面方式： 计算机(右键)->管理 ，然后就可以打开“计算机管理”界面，在界面的左侧有一个系统工具->任务计划程序 ，在此我们可以创建定时执行的计划任务，并且可以为每个任务添加触发器和具体的操作。

;

具体的使用方法应该很简单，这里就不再一一介绍。

CMD 命令行方式：主要命令是 SchTasks (另外还有 AT 命令，该命令用来调用或者执行已经存在于 Task Schedule 列表中的 Task)．命令行形式可以用于制作安装和部署包。如果用界面方式的话，安装和部署都不太方便。

SchTasks /?: 查看 SchTasks 命令，

数据库项目文档

SCHTASKS /parameter [arguments]

描述:

允许管理员创建、删除、查询、更改、运行和中止本地或远程系统上的计划任务。

参数列表:

/Create　　创建新计划任务。

/Delete　　删除计划任务。

/Query　　显示所有计划任务。

/Change　　更改计划任务属性。

/Run　　按需运行计划任务。

/End　　中止当前正在运行的计划任务。

/ShowSid　　显示与计划的任务名称相应的安全标识符。

/?　　显示此帮助消息

Windows 任务计划程序需要：具体的任务执行程序(exe,批处理)；而通过设置，让 Windows 来控制具体的执行的具体时间，重复度，触发条件，等。我们可以控制的是 具体的任务执行程序。 当然如果我们将什么时候定时执行任务，以及如何重复执行，触发条件等都写在程序里，也未尝不可，但是这样的话，就抹杀了 Windows 任务计划程序的特有功能了！！

接下来，我们来看一下，Windows Service 如何解决自动定时执行任务。

数据库项目文档

利用 Windows Service 解决自动定时执行任务的方案是，制作一个 Windows Service 程序，在 Service 内部部署好具体的任务程序，以及控制任务什么时候执行，如何重复，等等。他的可控性和自定义空间相比任务计划程序大，但是自己的工作量相对而言就大一些。

有关如何制作 Windows Service 程序，我在下篇文章中会介绍，这里只介绍如何部署和卸载。

部署 Windows Service 只能通过命令行形式来部署。但是具体的命令有两种: *InstallUtil* 和 *sc*

（也可以在 Windows Service 的 Program 启动点中添加自定义的 Installer 和启动参数来实现不用 InstallUtil 和 sc 来安装和卸载，最后还是要通过命令行形式输入运行参数来安装，这个我会在后续的文章中详解如何在 Windows Service 中添加启动参数和自定义 Installer）

InstallUtil.exe 是.NetFramework 自带的工具，一般他的路径在 C:\Windows\Microsoft.NET\Framework*version*\ 下。

使用方法： installutil [/uninstall][option [...]]assemblyname

其中 assemblyname 必须使用程序集的版本、区域性和公钥标记完全限定程序集名称

详情请见 InstallUtil.exe (MSDN)

SC 是系统自带的命令，不需要.Net Framework（当然如果希望运行 C#程序，.Net Framework 是必须的，所以一般情况下我们都可以用 InstallUtil 命令来部署和卸载）。我以前做开发 Service 时碰到的一个问题是，当我用 InstallUtil 部署好一个 Service，但由于程序更新，需要卸载后重新部署，这个时候，就出现用 InstallUitl /u 无法卸载的情况，具体的原因还不太明白，但是 MSDN 上说明了 InstallUtil 安装的时候是启用事务的，但是卸载的时候不启用事务，会不会是这个原因呢？？ 但是，这中情况下，我却可以用 sc delete 命令来卸载。

有关 sc 命令的使用方法，sc /?:

数据库项目文档

描述:

SC 是用于与服务控制管理器和服务进行通信的命令行程序。

用法:

sc <server> [command] [service name] <option1> <option2>...

选项 <server> 的格式为“\\servername”

键入 "sc [command]" 可以获得有关命令的进一步帮助

命令:

query----------查询服务的状态，

或枚举服务类型的状态。

queryex---------查询服务的扩展状态，

或枚举服务类型的状态。

start----------启动服务。

pause----------向服务发送 PAUSE 控制请求。

interrogate-----向服务发送 INTERROGATE 控制请求。

continue--------向服务发送 CONTINUE 控制请求。

stop----------向服务发送 STOP 请求。

config----------更改服务的配置(永久)。

description-----更改服务的描述。

failure---------更改服务失败时执行的操作。

failureflag-----更改服务的失败操作标志。

sidtype---------更改服务的服务 SID 类型。

privs----------更改服务的所需权限。

qc----------查询服务的配置信息。

qdescription----查询服务的描述。

qfailure--------查询失败时服务执行的操作。

qfailureflag----查询服务的失败操作标志。

qsidtype--------查询服务的服务 SID 类型。

qprivs----------查询服务的所需权限。

qtriggerinfo----查询服务的触发器参数。

qpreferrednode--查询首选的服务 NUMA 节点。

delete----------(从注册表)删除服务。

create----------创建服务(将其添加到注册表)。

control---------向服务发送控制。

sdshow----------显示服务的安全描述符。

sdset-----------设置服务的安全描述符。

showsid---------显示相应于假定名称的 SID 字符串。

triggerinfo-----配置服务的触发器参数。

preferrednode---设置首选的服务 NUMA 节点。

GetDisplayName--获取服务的 DisplayName

GetKeyName------获取服务的 ServiceKeyName。

EnumDepend------枚举服务的依存关系。

下列命令不要求服务名称:

sc <server> <command> <option>

boot------------(ok | bad) 指示是否将上一次启动保存为

最近一次已知的正确启动配置

Lock------------锁定服务数据库

QueryLock-------查询 SCManager 数据库的 LockStatus

示例:

sc start MyService

是否想参阅 QUERY 和 QUERYEX 命令的帮助? [y | n]:

y

QUERY 和 QUERYEX 选项:

如果查询命令带服务名称，将返回
该服务的状态。其他选项不适合这种
情况。如果查询命令不带参数或
带下列选项之一，将枚举此服务。

type= 要枚举的服务的类型(driver, service, all)
默认 = service)

state= 要枚举的服务的状态 (inactive, all)
(默认 = active)

bufsize= 枚举缓冲区的大小(以字节计)
(默认 = 4096)

ri= 开始枚举的恢复索引号
(默认 = 0)

group= 要枚举的服务组
(默认 = all groups)

语法示例

sc query - 枚举活动服务和驱动程序的状态

sc query eventlog - 显示 eventlog 服务的状态

sc queryex eventlog - 显示 eventlog 服务的扩展状态

sc query type= driver - 仅枚举活动驱动程序

sc query type= service - 仅枚举 Win32 服务

sc query state= all - 枚举所有服务和驱动程序

sc query bufsize= 50 - 枚举缓冲区为 50 字节

sc query ri= 14 - 枚举时恢复索引 = 14

sc queryex group= "" - 枚举不在组内的活动服务

sc query type= interact - 枚举所有不活动服务

sc query type= driver group= NDIS - 枚举所有 NDIS 驱动程序

上面的用法都是来自系统 help，具体使用很简单，试一试就 OK 了。

Windows Service 是一种完全自定义控制的实现自动定时执行任务的方法，可以自定义的东西很多，灵活性很高，所以使用起来可能会比 Windows 任务计划程序稍微复杂一些，适合灵活度和扩张性要求高的情况。

最后，我们简单介绍一些 SQL Agent 中 Job。

SQL Agent 是 SQL Server 的一个服务之一，用之前，我们需要启动 SQL Agent Windows Service. 这个只适用于数据库方面的数据定时更新，定时发送邮件，等等与数据库相关任务，但是牵涉到桌面应用方面的可能就无能为力了，这里不再多介绍了！

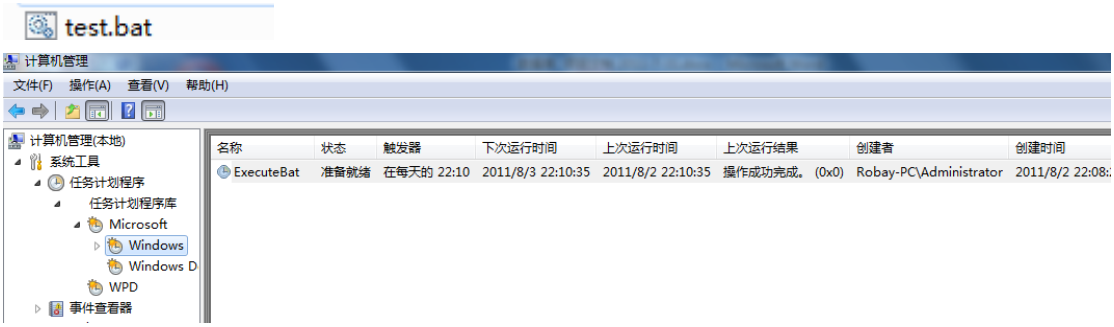
总而言之，使用何种方法来实现任务的定时的按计划执行 ，需要根据具体的情况来定：

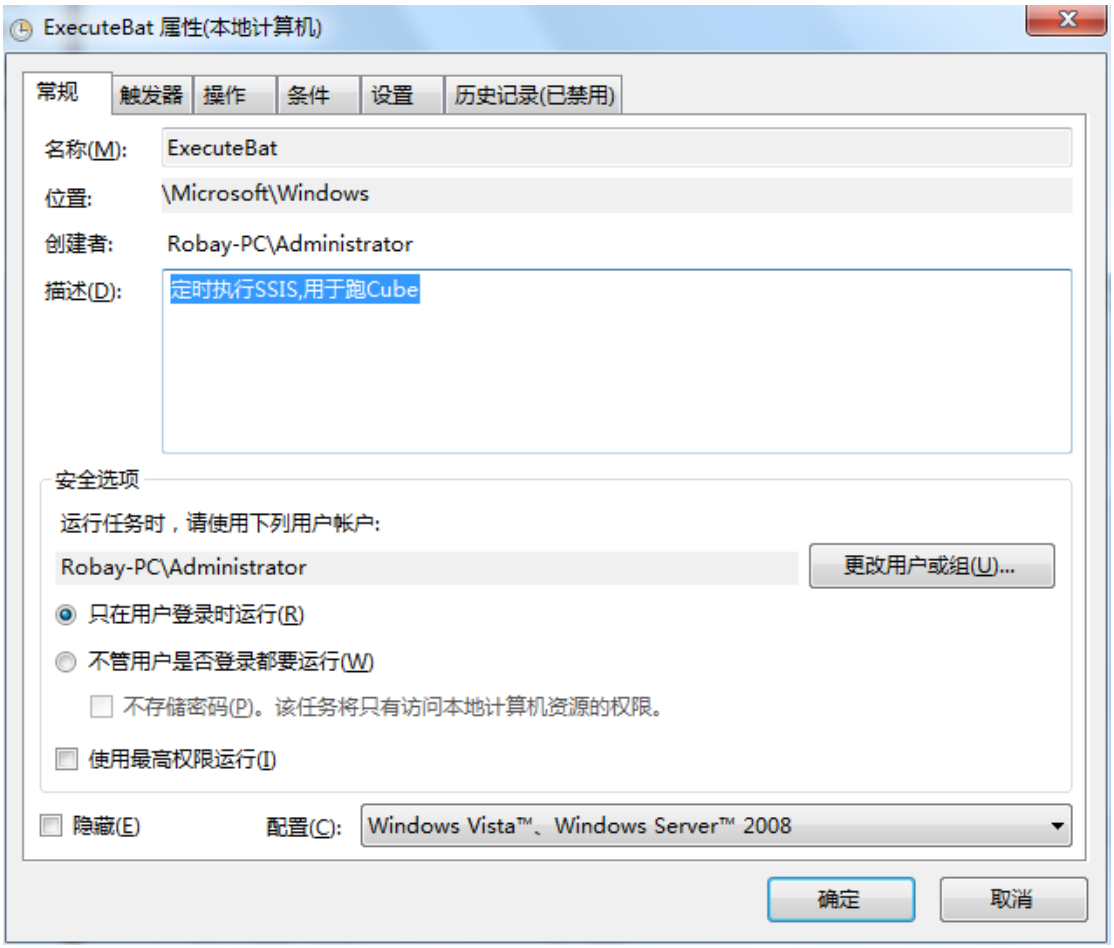
- 仅仅与数据库相关的，用 SQL Agent
- 不需要考虑太多灵活性和扩展性的，用 Windows 任务计划程序
- 灵活度高，需要考虑扩展应用（必须添加 Job，按照自定义方式执行 Job,... ...）,那我们可以选择 Windows Service

接下来的文章，我会用一个比较简单的例子，介绍如何构建，部署 Windows Service！

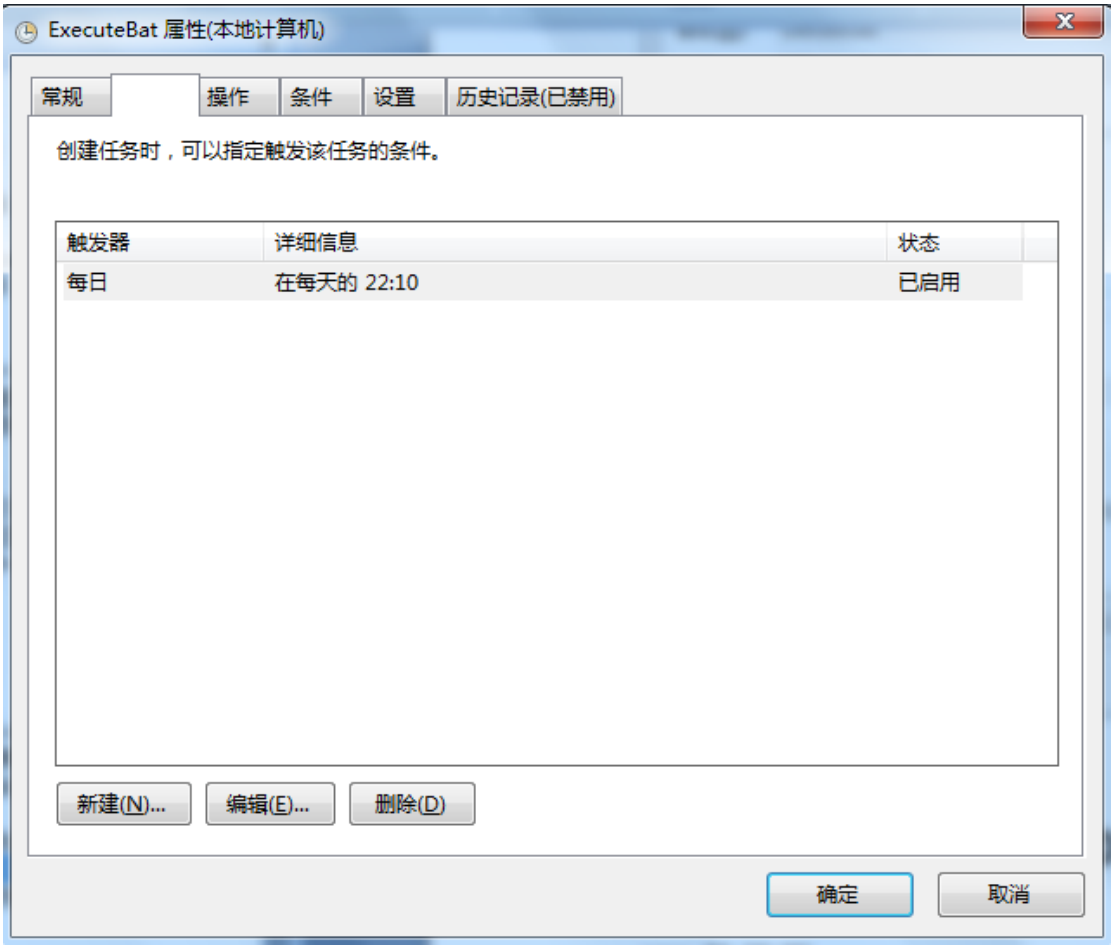
Bat 执行 SSIS 中的 DTS 包,并放到任务计划程序里面每天定时执行如下

```
  cd C:\Program Files\Microsoft SQL Server\90\DTS\Binn
dtexec /file "d:\Package.dtsx"> c:\testlog.log
```

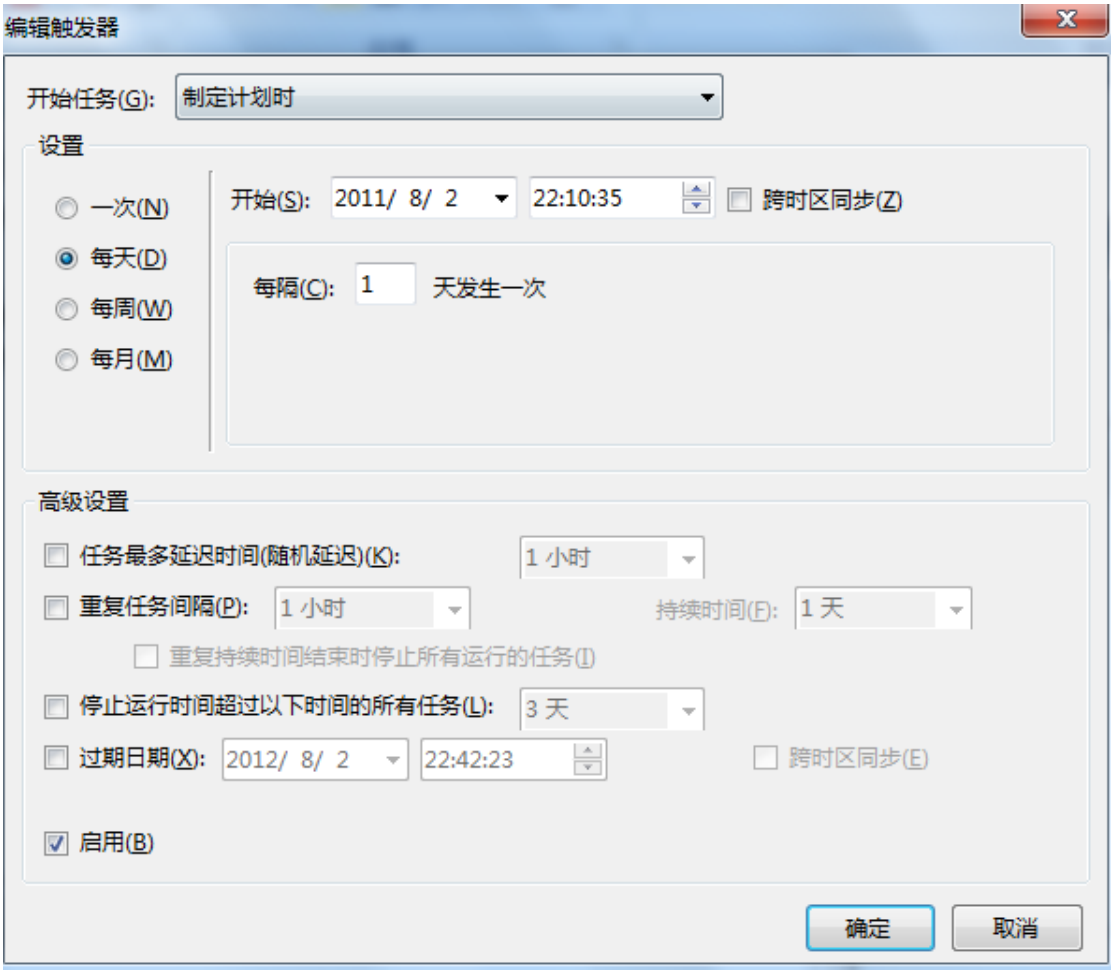




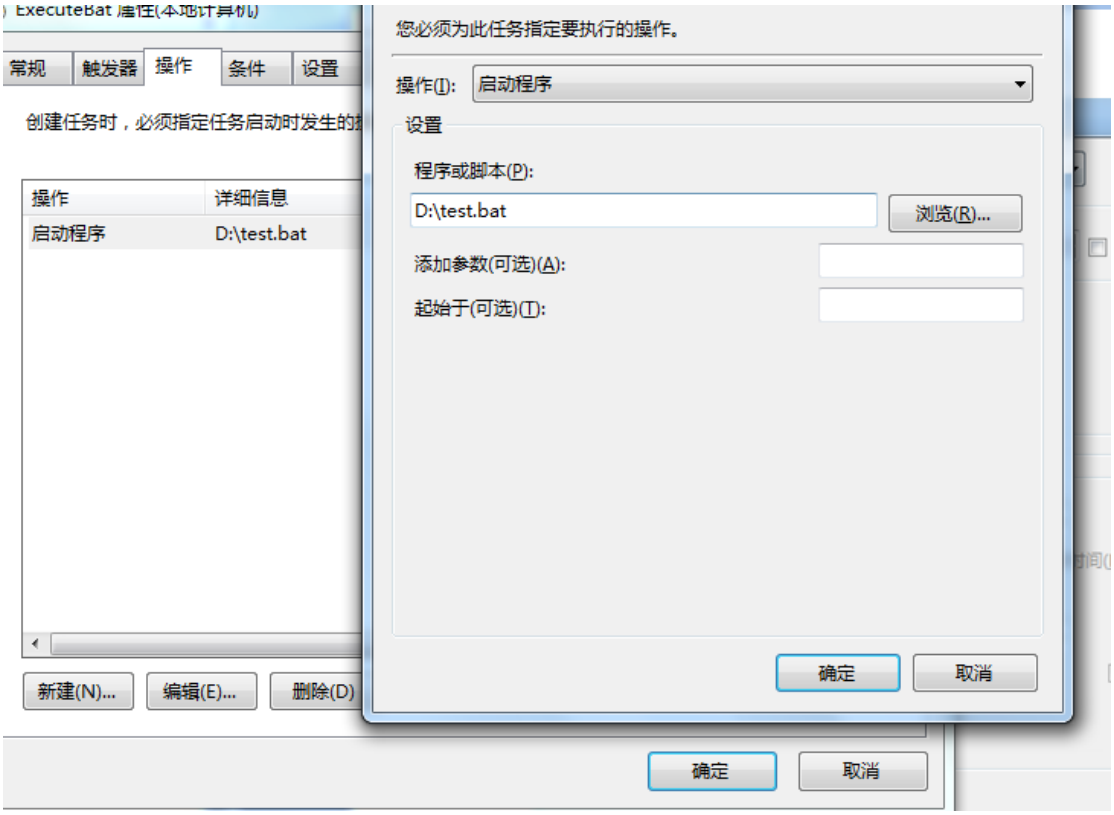
数据库项目文档



数据库项目文档



数据库项目文档



聚集索引和非聚集索引的区别

使用聚集索引

聚集索引确定表中数据的物理顺序。聚集索引类似于电话簿，后者按姓氏排列数据。由于聚集索引规定数据在表中的物理存储顺序，因此一个表只能包含一个聚集索引。但该索引可以包含多个列（组合索引），就像电话簿按姓氏和名字进行组织一样。

聚集索引对于那些经常要搜索范围值的列特别有效。使用聚集索引找到包含第一个值的行后，便可以确保包含后续索引值的行在物理相邻。例如，如果应用程序执行的一个查询经常检索某一日期范围内的记录，则使用聚集索引可以迅速找到包含开始日期的行，然后检索表中所有相邻的行，直到到达结束日期。这样有助于提高此类查询的性能。同样，如果对从表中检索的数据进行排序时经常要用到某一列，则可以将该表在该列上聚集（物理排序），避免每次查询该列时都进行排序，从而节省成本。

当索引值唯一时，使用聚集索引查找特定的行也很有效率。例如，使用唯一雇员 ID 列 emp_id 查找特定雇员的最快速的方法，是在 emp_id 列上创建聚集索引或 PRIMARY KEY 约束

数据库项目文档

使用非聚集索引
非聚集索引与课本中的索引类似。数据存储在一个地方，索引存储在另一个地方，索引带有指针指向数据的存储位置。索引中的项目按索引键值的顺序存储，而表中的信息按另一种顺序存储（这可以由聚集索引规定）。如果在表中未创建聚集索引，则无法保证这些行具有任何特定的顺序。

与使用书中索引的方式相似，Microsoft® SQL Server™ 2000 在搜索数据值时，先对非聚集索引进行搜索，找到数据值在表中的位置，然后从该位置直接检索数据。这使非聚集索引成为精确匹配查询的最佳方法，因为索引包含描述查询所搜索的数据值在表中的精确位置的条目。如果基础表使用聚集索引排序，则该位置为聚集键值；否则，该位置为包含行的文件号、页号和槽号的行 ID (RID)。例如，对于在 emp_id 列上有非聚集索引的表，如要搜索其雇员 ID (emp_id)，SQL Server 会在索引中查找这样一个条目，该条目精确列出匹配的 emp_id 列在表中的页和行，然后直接转到该页该行。

聚集索引就是将数据记录直接保存在此索引所建立的 B+树中。

非聚集索引则只在 B+树中保存记录的位置信息。

以下为 copy：
汉语字典的正文本身就是一个聚集索引。比如，我们要查“安”字，就会很自然地翻开字典的前几页，因为“安”的拼音是“an”，而按照拼音排序汉字的字典是以英文字母“a”开头并以“z”结尾的，那么“安”字就自然地排在字典的前部。如果您翻完了所有以“a”开头的部分仍然找不到这个字，那么就说明您的字典中没有这个字；同样的，如果查“张”字，那您也会将您的字典翻到最后部分，因为“张”的拼音是“zhang”。也就是说，字典的正文部分本身就是一个目录，您不需要再去查其他目录来找到您需要找的内容。正文内容本身就是一种按照一定规则排列的目录称为“聚集索引”。
如果您认识某个字，您可以快速地从自动中查到这个字。但您也可能会遇到您不认识的字，不知道它的发音，这时候，您就不能按照刚才的方法找到您要查的字，而需要去根据“偏旁部首”查到您要找的字，然后根据这个字后的页码直接翻到某页来找到您要找的字。但您结合“部首目录”和“检字表”而查到的字的排序并不是真正的正文的排序方法，比如您查“张”字，我们可以看到在查部首之后的检字表中“张”的页码是 672 页，检字表中“张”的上面是“驰”字，但页码却是 63 页，“张”的下面是“弩”字，页面是 390 页。很显然，这些字并不是真正的分别位于“张”字的上下方，现在您看到的连续的“驰、张、弩”三字实际上就是他们在非聚集索引中的排序，是字典正文中的字在非聚集索引中的映射。我们可以通过这种方式来找到您所需要的字，但它需要两个过程，先找到目录中的结果，然后再翻到您所需要的页码。

我们把这种目录纯粹是目录，正文纯粹是正文的排序方式称为“非聚集索引”。
主键 就是一种聚集索引 和 你打开数据库的排序 相同
非聚集索引 就像你把日期 当为索引 排出的顺序和你 直接打开表的顺序不同

聚集索引就是按照索引排列物理存储位置。

聚集索引的查询性能好于非聚集索引，但是维护代价很大，对于他的数据改变会引起整行数据的物理位置移动。同时聚集索引还要为非聚集索引提供索引服务，所以尽量不用过大的列或过多的列作聚集索引

聚集索引可以极大优化大于，小于，group by 和 order by 以及 join 语句的查询性

一张表只能由一个聚集索引
聚集索引:物理存储按照索引排序
非聚集索引:物理存储不按照索引排序
聚集索引：插入数据时速度要慢
查询数据比非聚集数据的速度快

SQL 索引

索引是以表列为基础的数据库对象。索引中保存着表中排序的索引列，并且纪录了索引列在数据库表中的物理存储位置，实现了表中数据的逻辑排序。通过索引，可以加快数据的查询速度和减少系统的响应时间;可以使表和表之间的连接速度加快。

但是，不是在任何时候使用索引都能够达到这种效果。若在不恰当的场合下，使用索引反而会事与愿违。所以，在 SQL Server 数据库中使用索引的话，还是需要遵守一定的规则。笔者觉得，主要是需要遵守六大铁律。

铁律一：天下没有免费的午餐，使用索引是需要付出代价的。

索引的优点有目共睹，但是，却很少有人关心过采用索引所需要付出的成本。若数据库管理员能够对索引所需要付出的代价有一个充分的认识，也就不会那么随意到处建立索引了。

仔细数数，其实建立索引的代价还是蛮大的。如创建索引和维护索引都需要花费时间与精力。特别是在数据库设计的时候，数据库管理员为表中的哪些字段需要建立索引，要调研、要协调。如当建有索引的表中的纪录又增加、删除、修改操作时，数据库要对索引进行重新调整。虽然这个工作数据库自动会完成，但是，需要消耗服务器的资源。当表中的数据越多，这个消耗的资源也就越多。如索引是数据库中实际存在的对象，所以，每个索引都会占用一定的物理空间。若索引多了，不但会占用大量的物理空间，而且，也会影响到整个数据库的运行性能。

可见，数据库管理员若要采用索引来提高系统的性能，自身仍然需要付出不少的代价。数据库管理员现在要考虑的就是如何在这两个之间取得一个均衡。或者说，找到一个回报与投入的临界点。

铁律二：对于查询中很少涉及的列或者重复值比较多的列，不要建立索引。

在查询的时候，如果我们不按某个字段去查询，则在这个字段上建立索引也是浪费。如现在有一张员工信息表，我们可能按员工编号、员工姓名、或者出身地去查询员工信息。但是，我们往往不会按照身份证号码去查询。虽然这个身份证号码是唯一的。此时，即使在这个字段上建立索引，也不能够提高查询的速度。相反，增加了系统维护时间和占用了系统空间。这简直就是搬起石头砸自己的脚呀。

另外，如上面的员工信息表，有些字段重复值比较多。如性别字段主要就是“男”、“女”；职位字段中也是有限的几个内容。此时，在这些字段上添加索引也不会显著的增加查询速度，减少用户响应时间。相反，因为需要占用空间，反而会降低数据库的整体性能。

数据库索引管理中的第二条铁律就是，对于查询中很少涉及的列或者重复值比较多的列，不要建立索引。

铁律三：对于按范围查询的列，最好建立索引。

在信息化管理系统中，很多时候需要按范围来查询某些交易记录。如在 ERP 系统中，经常需要查询当月的销售订单与销售出货情况，这就需要按日期范围来查询交易记录。如有时候发现库存不对时，也需要某段时期的库存进出情况，如 5 月 1 日到 12 月 3 日的库存交易情况等等。此时，也是根据日期来进行查询。

对于这些需要在指定范围内快速或者频繁查询的数据列，需要为其建立索引。因为索引已经排序，其保存的时候指定的范围是连续的，查询可以利用索引的排序，加快查询时间，减少用户等待时间。

不过，若虽然可能需要按范围来进行查询，但是，若这个范围查询条件利用的不多的情况下，最好不好采用索引。如在员工信息表中，可能需要查询 2008 年 3 月份以前入职的员工明细，要为他们增加福利。但是，由于表中记录不多，而且，也很少进行类似的查询。若维这个字段建立索引，虽然无伤大雅，但是很明显，索引所获得的收益要低于其成本支出。对数据库管理员来说，是得不偿失的。

再者，若采用范围查询的话，最好能利用 TOP 关键字来限制一次查询的结果。如第一次按顺序只显示前面的 500 条记录等等。把 TOP 关键字跟范围一起使用，可以大大的提高查询的效率。

铁律四：表中若有主键或者外键，一定要为其建立索引。

定义有主键的索引列，一定要为其建立索引。因为主键可以加速定位到表中的某一行。结合索引的作用，可以使得查询的速度加倍。如在员工信息表中，我们往往把员工编号设置为主键。因为这不但可以提高查询的速度，而且因为主键要求记录的唯一，还可以保证员工编号的唯一性。

此时，若再把这个员工编号字段设置为索引，则通过员工编号来查询员工信息，其效率要比没有建立索引高出许多。

另外，若要使得某个字段的值唯一，可以通过两种索引方式实现。一种就是上面所讲的主键索引。还有一种就是唯一索引，利用 UNIQUE 关键字指定字段内容的唯一性。这两种方式都会在表中的指定列上自动创建唯一索引。这两种方式的结果没有明显的区别。查询优化器不会区分到底是哪种方式建立的唯一性索引，而且他们进行数据查询的方式也是相同的。

若某张表中的数据列定义有外键，则最好也要为这个字段建立索引。因为外键的主要作用就在于表与表之间的连接查询。若在外键上建立索引，可以加速表与表之间的连接查询。如在员工基本信息表中，有一个字段为员工职位。由于员工职位经常在变化，在这里，存储的其实只是一个员工职位的代码。在另外一张职位信息表中详细记录着该职位的相关信息。此时，这个员工职位字段就是外键。若在这个字段上建立外键，则可以显著提高两张表的连接速度。而且，记录越多，其效果越加明显。

所以，当表中有外键或者主键的时候，就最好为其建立索引。通过索引，可以强化主键与外键的作用，提高数据库的性能。

铁律五：对于一些特殊的数据类型，不要建立索引。

在表中，有些字段比较特殊。如文本字段(TXT)、图像类型字段(IMAGE)等等。如果表中的字段属于这些数据类型，则最好不要为其建立索引。因为这些字段有一些共同的特点。如长度不确定，要么很长，几个字符;要么就是空字符串。如文本数据类型常在应用系统的数据库表中用来做备注的数据类型。有时候备注很长，但有时候又没有数据。若这种类型的字段上建立索引，那根本起不了作用。相反，还增加了系统的负担。

所以，在一些比较特殊的数据类型上，建立索引要谨慎。在通常情况下，没有必要为其建立索引。但是，也有特殊的情况。如有时候，在 ERP 系统中，有产品信息这个表，其中有个产品规格这个字段。有时候，其长度可能长达 5000 个字符。此时，只有文本型的数据类型可以容纳这么大的数据量。而且，在查询的时候，用户又喜欢通过规格这个参数来查询产品信息。此时，若不为这个字段建立索引的话，则查询的速度会很慢。遇到这种情况时，数据库管理员只有牺牲一点系统资源，为其建立索引。

从这里也可以看出，虽然以上几条说的时铁律，但是，是否需要遵循，还是需要数据库管理员根据企业的实际情况，做出合理的选择。

铁律六：索引可以跟 Where 语句的集合融为一体。

用户在查询信息的时候，有时会经常会用到一些限制语句。如在查询销售订单的时候，经常

会用到客户以及下单日期的条件集合；如在查询某个产品的库存交易情况时，就会利用产品编号与交易日期起止日期的条件集合。

对于这些经常用在 Where 子句中的数据列，将索引建立在 Where 子句的集合过程中，对于需要加速或者频繁检索的数据列，可以让这些经常参与查询的数据列按照索引的排序进行查询，以加快查询的时间。

总之，索引就好像一把双刃剑，即可以提高数据库的性能，也可能对数据库的性能起到反面作用。作为数据库管理员，要有这个能力判断在合适的时间、合适的业务、合适的字段上建立合适的索引。以上六个铁律，只是对建立索引的一些基本要求

SQL 临时表

临时表其实是放在数据库 tempdb 里的一个用户表
分两种：
一种是以#(局部)或##(全局)开头的表，这种表在会话期间存，会话结束则自动删除；
另一种，如果创建时不以#或##开头，而用 tempdb.TempTable 来命名它，则该表可在数据库重启前一直存在.
以上两种都可手动用
drop table TempTableName 来删除.

临时表其实是放在数据库 tempdb 里的一个用户表
分两种：
一种是以#(局部)或##(全局)开头的表，这种表在会话期间存，会话结束则自动删除；
另一种，如果创建时不以#或##开头，而用 tempdb.TempTable 来命名它，则该表可在数据库重启前一直存在.
以上两种都可手动用
drop table TempTableName 来删除.

排序

```
select max(列) from tb group by left(列,6)
```

SQL 查询出来的结果转化为.txt 文件

```
--
--EXEC master..xp_cmdshell 'bcp "select * from City.dbo.table1"
--queryout c:\mb.txt -c -t,  -Usa -P123'


if OBJECT_ID(N'Tbl1',N'U') is not null drop table Tbl1
go
create table Tbl1
(
  ErrorMessage nvarchar(max)
)
go
declare @date varchar(50),@sql varchar(200),@IntPar int;
declare @table table (ErrorMessage nvarchar(max))
select
@date=convert(varchar(10),getdate(),120),@IntPar=cast(replace(convert
(date,getdate()),'-','') as int);
insert into Tbl1
select a.Message from msdb.dbo.sysjobhistory  a left join
msdb.dbo.sysjobs b
on a.job_id=b.job_id where b.name='exec'
--and  run_date= @IntPar

set @sql='bcp Tbl1 out c:\'+@date+'.txt -c -SRobay-PC -Usa -P123';
print @sql

EXEC master..xp_cmdshell @sql
```

SQL 数据更新原理

```
--> 测试数据：[ta]
if object_id('[ta]') is not null drop table [ta]
go
create table [ta]([姓名] varchar(4),[数量] sql_variant)
insert [ta]
```

```
select '张三',null union all
select '李四',null union all
select '王五',null
--> 测试数据：[tb]
if object_id('[tb]') is not null drop table [tb]
go
create table [tb]([姓名] varchar(4))
insert [tb]
select '张三' union all
select '张三' union all
select '李四' union all
select '王五' union all
select '王五' union all
select '王五'
--------------开始查询--------------------------
update
  a
set
  数量=b.数量
from
  ta a,
  (select 姓名,COUNT(1) as 数量from tb group by 姓名)b
where
   a.姓名=b.姓名

select * from ta
```

SQL 事物的讲解

```
begin  tran  ok  --开始一个事务OK
 delete  from  rxqz  where qz=   'rx015 ' --删除数据
save tran  bcd   --保存一个事务点命名为bcd
 update  sz  set   name='李丽s' where name= '李丽'--修改数据
if  @@error=0  --判断修改数据有没有出错
begin --如果出错
 rollback  tran  bcd  -- 回滚事务到BCD 的还原点
commit  tran  ok  --提交事务
end
else  --出错
```

```
commit  tran ok --提交事务
```

SQL 字符串截取问题，重点 CharIndex 的使用

```
declare @str varchar(20);
set @str='(100小时分)';
select charindex('小时',@str)
 select substring(@str,2,charindex('小时',@str)-2)


/*请教SQL Server2005中截取字符串问题?

有一个字段内容如下:

ASB-34A1-P033
AB-S2-D3
ABC-S32-D34
KJ-K4-ER-JK
H5-G323-FSS-G54
...

要求得到如下第二个'-'符号前面的数字如下:
1
2
32
4
323
*/
create table tb(name varchar(20))
insert into tb
select 'ASB-34A1-P033' union
select 'AB-S2-D3' union
select 'ABC-S32-D34' union
select 'H5-G323-FSS-G54'


select
substring(stuff(name,1,charindex('-',name),''),charindex('-',stuff(na
me,1,charindex('-',name),''))-1,1)
```

```
from tb
where len(name)-len(replace(name,'-','')) >= 2

select
REVERSE(left(REVERSE(left(RIGHT(name,len(name)-CHARINDEX('-',name)),
    CHARINDEX('-',RIGHT(name,len(name)-CHARINDEX('-',name)))-1)),

PATINDEX('%[^0-9]%',REVERSE(left(RIGHT(name,len(name)-CHARINDEX('-',n
ame)),

CHARINDEX('-',RIGHT(name,len(name)-CHARINDEX('-',name)))-1)))-1))
from tb
```

临时表和表变量区别

```
declare @tb table(id int,name varchar(50),age int) --创建表变量

insert @tb select 1,'nn',14
select * from @tb

create table #t(id int,name varchar(50),years int,nums int)--创建临时表

insert #t select 1,'nn',14,15
union all select 1,'nn',14,15
insert into #t  exec sp_gets  --可以用于存储过程或动态SQL结合

select * from #t
drop table #t --删除临时表

/*
```

当创建本地或全局临时表时，CREATE TABLE 语法支持除FOREIGN KEY 约束以外的其它所有约束定义。如果在临时表中指定FOREIGN KEY 约束，该语句将返回警告信息，指出此约束已被忽略，表仍会创建，但不具有FOREIGN KEY 约束。在FOREIGN KEY 约束中不能引用临时表。

考虑使用表变量而不使用临时表。当需要在临时表上显式地创建索引时，或多个存储过程或函数需要使用表值时，临时表很有用。通常，表变量提供更有效的查询处理。有关更多信息，

1．表变量的操作并不记录在transaction log里，所以对表变量不能用rollback，而临时表则可以．
2．任何有临时表的过程都不能被预编译(pre-compiled)，但是有表变量的过程的execution plan却有可能被预先生成．所以从这个意义来看，表变量的过程要比临时表的过程速度要快．
3．表变量跟普通变量一样，在内层的存储过程和exec(string)过程中是不可见的(临时变量则可以，几个有个大虾还专门发贴庆贺过这件事)，所以表变量也不能用在insert/exec语句里．
*/

Delete 和 Truncate 的区别

Delte 删除的时候记录日志，而truncate 不记录日志。

BI 中的报表和业务系统中的报表的区别

在 BI 实施过程中经常会遇到，做的报表业务系统也存在，而且在实施的测试阶段是拿业务系统的报表来验证 BI 中的报表。

首先我要说明的是，看业务需求是什么样的，如果是向领导汇报（周报、日报等），这种做法是十分正确的，但如果是分析决策或改进管理为目的的话，那么这种做法就是掩耳盗铃。一句话无用功。大家想想，分析准确性的保证是什么，一定是数据的准确并有效。业务数据包含了各种真实发生的情况数据，当然是准确的，但并不是全部有效地。所以 BI 中才叫 ETL 过程，而不叫数据迁移。

因此 BI 中的报表与业务系统中的报表第一个区别就是：BI 中的报表数据对分析来说是准确、有效地。

第二个区别就是 BI 教会你如何看报表。听起来让人好笑，那我就详细解释一下吧，就拿资产负债表来说，不同的岗位的人，看到该表后得出的结论可能存在不同，因为他们出发点、关注点不同，所以会得出不同的结论。因此三人行必有我师，在 BI 系统中，通过报表帮助可以教会你从各个角度看报表。从而提高你的管理水平，一致达到准确决策的目的。（千万不要告诉我业务系

统也可以做到，当然可以做到，IT 无所不能，只要有规则，但你能实施成功吗？去该业务系统。）

第三个区别就是知识体系。普通à优秀à卓越，差距在哪里呢，知识的宏观把控是一项非常重要的内容吧。因此，BI 中的报表与报表间，已经形成了知识体系，达到了宏观的高度，即形成了知识体系。

举例说明：我们在看多张报表时候，一定有一定顺序的看，当一张数据有问题，我们会找相关的报表查找。这个过程就是一个小小的知识体系。

第三个区别 BI 报表具有 BI 功能

…… 还有很多地方不同，这里就不一一介绍了。

方法一：分析模型现场演示

以 BO 或 share point 中的 excel service 为例，都提供了相应的工具，非常方便制作，可以事先做好例子。然后现场输入数据（也可体现输入），进行模拟演示。起到沙盘的作用。一般客户都会被高度吸引。

方法二：定制演示

这里指的不是拖拖拽拽。而是个性化定制，最好的例子就是 KPI 的个性化定制演示。这也是客户最关心的，可以随意方便的定制（度量和纬度范围）。

方法三：系统集成

不要等客户提出这个问题，而是主动提出。我在做售前过程中，基本每次都被问到（客户经常是一个系统一个用户名一个密码，对老大们来说，不断地在不同系统间输入用户名和密码是种折磨），后来我就主动说了。

SQL2005 执行缓存的效率

A:简单说一下：就是 MSSQL 对一些可以重复使用的过程、语句进行存储，减少相同过程、语句的重新编译（PS:语法、语义、执行计划等的分析）。那么执行缓存其实就是（记忆在大脑中）的这个东西，一旦 MSSQL，发现有相同的存储过程、或是 SQL 语句，它就可以直接运行，而不需要在进行语义、语法分析等。
Microsoft SQL Server 提供了一项强大的功能称为在 RAM 中的 tempdb。这使临时数据库 tempdb，这用于在排序中的工作空间和在某些联接操作创建临时表进行完全内存驻留。在某些特定的情况下，这可以提供性能优势。但是，如果不适当地使用，则在内存中的 tempdb，它可以消耗 SQL Server 高速缓存缓冲区系统，使用的内存，这可能会损害性能。 使用 SQL 缓存依赖技术,可以实现当应用程序中对应的数据库中的数据发生改变时

才清除应用程序中对应的缓存,最大化的提高程序的性能。

Tempdb 数据库记录了所有的临时表格，临时数据和临时创建的存储过程。它在每次服务重新启动的时候就会重新生成

不使用排名函数，求排名前两名的值

```
create table tb(id int,iorder int)
insert into tb
select 1,2 union all
select 1,3 union all
select 1,6 union all
select 1,6 union all
select 2,3 union all
select 2,1 union all
select 2,4
go

select *
from tb t
where (select count(*) from tb where id = t.id and iorder <= t.iorder) >
2
```

不同服务器数据库之间的数据操作

```
/*不同服务器数据库之间的数据操作*/

--创建链接服务器
exec sp_addlinkedserver   'ITSV ', ' ', 'SQLOLEDB ', '远程服务器名或 ip 地址 '
exec sp_addlinkedsrvlogin  'ITSV ', 'false ',null, '用户名 ', '密码 '

--查询示例
select * from ITSV.数据库名.dbo.表名

--导入示例
select * into 表 from ITSV.数据库名.dbo.表名

--以后不再使用时删除链接服务器
exec sp_dropserver  'ITSV ', 'droplogins '

--连接远程/局域网数据(openrowset/openquery/opendatasource)
--1、openrowset

--查询示例
select * from openrowset( 'SQLOLEDB ', 'sql 服务器名 '; '用户名 '; '密码 ', 数据库名.dbo.表名)

--生成本地表
select * into 表 from openrowset( 'SQLOLEDB ', 'sql 服务器名 '; '用户名 '; '密码 ',数据库名.dbo.表名)

--把本地表导入远程表
insert openrowset( 'SQLOLEDB ', 'sql 服务器名 '; '用户名 '; '密码 ',数据库名.dbo.表名)
select *from 本地表

--更新本地表
update b
set b.列 A=a.列 A
 from openrowset( 'SQLOLEDB ', 'sql 服务器名 '; '用户名 '; '密码 ',数据库名.dbo.表名)as a inner join 本地表 b
```

```
on a.column1=b.column1

--openquery 用法需要创建一个连接

--首先创建一个连接创建链接服务器
exec sp_addlinkedserver   'ITSV ', ' ', 'SQLOLEDB ', '远程服务器名或 ip 地址 '
--查询
select *
FROM openquery(ITSV,  'SELECT *  FROM 数据库.dbo.表名 ')
--把本地表导入远程表
insert openquery(ITSV,  'SELECT *  FROM 数据库.dbo.表名 ')
select * from 本地表
--更新本地表
update b
set b.列 B=a.列 B
FROM openquery(ITSV,  'SELECT * FROM 数据库.dbo.表名 ') as a
inner join 本地表 b on a.列 A=b.列 A

--3、opendatasource/openrowset
SELECT   *
FROM   opendatasource( 'SQLOLEDB ',  'Data Source=ip/ServerName;User ID=登陆名;Password=密码 ' ).test.dbo.roy_ta
--把本地表导入远程表
```

不同服务器数据库之间的数据操作 2

```
EXEC sp_addlinkedserver
     @server='DBVIP',--被访问的服务器别名(任意的名称)
     @srvproduct='',
     @provider='SQLOLEDB',
     @datasrc='MYSQLServer'   --要访问的服务器(SQL SERVER实例名)

EXEC sp_addlinkedsrvlogin
    'DBVIP', --被访问的服务器别名
    'false',
    NULL,
```

```
    'sa', --登陆链接服务器的帐号
    '1q2w3e4R' --登陆链接服务器的密码

--查看已注册的链接服务器
--exec sp_linkedservers

--把本地的表数据插入到链接服务器上的表中
SELECT * INTO DBVIP.database_name.dbo.table_name FROM local_table_name
--DBVIP.database_name.dbo.table_name 远程服务器完整表名(必须用部分表示)
--local_table_name 本地表名

--用完后可删除
--Exec sp_droplinkedsrvlogin DBVIP,NULL --删除链接服务器的登陆帐户
--Exec sp_dropserver DBVIP --删除链接服务器
```

EXEC　Sp_executesql 函数的使用说明

sp_executesql提供接口
sp_executesql命令比EXEC命令更灵活，因为它提供一个接口，该接口及支持输入参数也支持输出参数。这功能使你可以创建带参数的查询字符串，这样就可以比EXEC更好的重用执行计划,sp_executesql的构成与存储过程非常相似，不同之处在于你是动态构建代码。它的构成包括：代码快，参数声明部分，参数赋值部分。说了这么多，还是看看它的语法吧

```
EXEC sp_executesql
@stmt = <statement>,--类似存储过程主体
@params = <params>, --类似存储过程参数部分
<params assignment> --类似存储过程调用
```

@stmt参数是输入的动态批处理,它可以引入输入参数或输出参数,和存储过程的主体语句一样,只不过它是动态的，而存储过程是静态的，不过你也可以在存储过程中使用sp_executesql;
@params参数与定义输入/输出参数的存储过程头类似，实际上和存储过程头的语法完全一样;
@<params assignment> 与调用存储过程的EXEC部分类似。
为了说明sp_executesql对执行计划的管理优于EXEC,我将使用前面讨论EXEC时用到的代码。

```
1:  DECLARE @TableName VARCHAR(50),@sql NVARCHAR(MAX),@OrderID INT;
2:  SET @TableName = 'Orders ';
3:  SET @OrderID = 10251;
4:  SET @sql = 'SELECT * FROM '+QUOTENAME(@TableName) + ' WHERE OrderID
= @OID ORDER BY ORDERID DESC'
5:  EXEC sp_executesql
6:      @stmt = @sql,
7:      @params = N'@OID AS INT ',
```

```
   8:      @OID = @OrderID
```

在调用该代码和检查它生成的执行计划前，先清空缓存中的执行计划；

```
DBCC FREEPROCCACHE
```

将上面的动态代码执行3次，每次执行都赋予@OrderID 不同的值，然后查询sys.syscacheobjects表，并注意它的输出，优化器只创建了一个备用计划，而且该计划被重用的3次

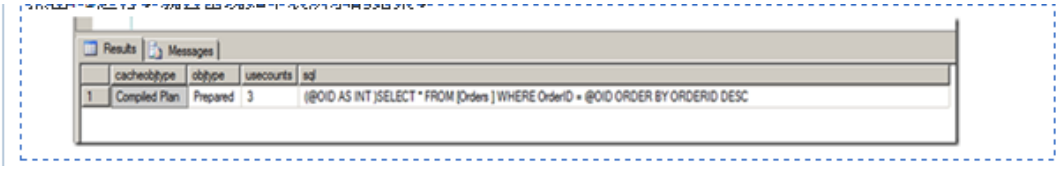
```
SELECT cacheobjtype,objtype,usecounts,sql FROM sys.syscacheobjects
WHERE sql NOT LIKE '%cache%' AND sql NOT LIKE '%sys.%' AND sql NOT LIKE
'%sp_executesql%'
```

点击F5运行，就会出现如下表所示的结果；

sq_executesql的另一个与其接口有关的强大功能是，你可以使用输出参数为调用批处理中的变量返回值。利用该功能可以避免用临时表返回数据，从而得到更高效的代码和更少的重新编译。定义和使用输出参数的语法与存储过程类似。也就是说，你需要在声明参数时指定OUTPUT子句。例如，下面的静态代码简单的演示了如何从动态批处理中利用输出参数@p把值返回到外部批处理中的变量@i.

```
DECLARE @sql AS NVARCHAR(12),@i AS INT;
SET @sql = N' SET @p = 10';
EXEC sp_executesql
    @stmt = @sql,
    @params = N'@p AS INT OUTPUT',
    @p = @i OUTPUT
SELECT @i
```

该代码返回输出10



sq_executesql 的另一个与其接口有关的强大功能是，你可以使用输出参数为调用批处理中的 变量返回值。利用该功能可以避免用临时表返回数据，从而得到更高效的代码和更少的重新编译。定义和使用输出参数的语法与存储过程类似。也就是说，你需要在声明参数时指定 OUTPUT 子句。例如，下面的静态代码简单的演示了如何从动态批处理中利用输出参数@p 把值返回到外部批处理中的变量@i.

```
DECLARE @sql AS NVARCHAR(12),@i AS INT;
SET @sql = N' SET @p = 10';
EXEC sp_executesql
    @stmt = @sql,
    @params = N'@p AS INT OUTPUT',
    @p = @i OUTPUT
```

```
SELECT @i
```

该代码返回输出 10

实战说明：【动态SQL使用说明】

```
exec sp_executesql N'EXEC Pro_PM_ML_StorAlarm @vTblName=@P1, @vBillCode=@P2 , @vWareHCode=@P3, @vBillDate=@P4, @iAlarmType=@P5 ,
@iAlarmLevel=@P6, @iBillKind=@P7
', N'@P1 varchar(8),@P2 varchar(9),@P3 varchar(2),@P4 varchar(10),@P5 int,@P6 int,@P7 smallint', 'MLLTZDMX', 'MJ2000015', 'X1', '2011-12-01', 0,
1, 1
 exec sp_executesql N'EXEC Pro_PM_ML_StorAlarm @vTblName=@P1, @vBillCode=@P2 , @vWareHCode=@P3, @vBillDate=@P4, @iAlarmType=@P5 ,
 @iAlarmLevel=@P6, @iBillKind=@P7
 ', N'@P1 varchar(8),@P2 varchar(9),@P3 varchar(2),@P4 varchar(10),@P5 int,@P6 int,@P7 smallint', 'MLLTZDMX', 'MJ2000015', 'X1', '2011-12-01', 0,
 1, 1
```

注意参数的类型，决定了以后输出的大小。

```
set @strsql='select @history_db=history_db from '+@tablename+' where djbh='''+@djbh+''''
EXEC sp_executesql @strsql ,N'@history_db varchar(4) out',@history_db_cur out ;


if(@history_db_cur is null)
begin
  SET @ErrorSave = -2
  RETURN @ErrorSave;
end
```

timestamp 与 binary 的关系说明

timestamp 这种数据类型表现自动生成的二进制数，确保这些数在数据库中是唯一的 binary 是二进制数据类型，可以认为，前者是后者的一个特例（即子集），只不过系统为你做了一部分工作。

timestamp 既然命名为时间戳，当然也记录了日期信息，只不过是以二进制方式存储的。但是它的值范围小于 datetime,也属于 datetime 的子集。

```sql
declare @n bigint
set @N=8001
select cast(@N as timestamp)

declare @d datetime
set @d=8001  -- 以1900-1-1为初始时间按dd去dateadd,8001dd.
select cast(@d as timestamp)

declare @x datetime
set @x='1921-11-28'
select cast(@x as binary(8))
```

SQL 中 xtype 的使用

syscolumns 表内的 xtype
数据库的 syscolumns 表里面.
name 就是列名,xtype 就是数据类型,但是这个 xtype 是数字的.
面是数字和数据类型对应的关系;
C = CHECK 约束
D = 默认值或 DEFAULT 约束
F = FOREIGN KEY 约束
L = 日志
FN = 标量函数
IF = 内嵌表函数
P = 存储过程
PK = PRIMARY KEY 约束（类型是 K）
RF = 复制筛选存储过程
S = 系统表
TF = 表函数
TR = 触发器
U = 用户表
UQ = UNIQUE 约束（类型是 K）
V = 视图
X = 扩展存储过程
xtype=34 'image'
xtype=35 'text'
xtype=36 'uniqueidentifier'
xtype=48 'tinyint'
xtype=52 'smallint'

```
xtype=56 'int'
xtype=58 'smalldatetime'
xtype=59 'real'
xtype=60 'money'
xtype=61 'datetime'
xtype=62 'float'
xtype=98 'sql_variant'
xtype=99 'ntext'
xtype=104 'bit'
xtype=106 'decimal'
xtype=108 'numeric'
xtype=122 'smallmoney'
xtype=127 'bigint'
xtype=165 'varbinary'
xtype=167 'varchar'
xtype=173 'binary'
xtype=175 'char'
xtype=189 'timestamp'   这个很重要，查找时候匹对的
xtype=231 'nvarchar'
xtype=239 'nchar'
xtype=241 'xml'
xtype=231 'sysname'

SELECT *
FROM   syscolumns c
      INNER JOIN sysobjects o
        ON c.id = o.id
           --AND o.name = 'DLLSD'
WHERE  c.xtype = 189
```

嵌套查询的使用

一个 select...From...Where 查询语句块可以嵌套在另一个 select...From...Where 查询块的 Where 子句
中，称为嵌套查询。外层查询称为父查询，主查询。内层查询称为子查询，从查询。
子查询可以嵌套多层，子查询查询到的结果又成为父查询的条件。子查询中不能有 order by 分组语句。
先处理子查询，再处理父查询。

1。 简单嵌套查询

查询选修课程号为'101'并且成绩高于学生号为'9501101'的所有学生的成绩.

```
select * from sclass
where cno='101' and degree>=
(select degree from sclass where sno='9501101'and cno='101')
select * from 成绩表
where 课程成绩<=
(select 课程成绩 from 成绩表 where 学号='20020001'and 课程代号='2002030002')
```

2。 带[not] in 的嵌套查询

查询有选修了课程的学生。

```
select sno,sname from student
where sno in(Select distinct sno from sclass)
```

查询没有选修了课程的学生。

```
select sno,sname from student
where sno not in(Select distinct sno from sclass)

select 学生表.学号,学生表.姓名,成绩表.课程成绩,成绩表.课程代号 from 学生表,成绩表
where 学生表.学号=成绩表.学号
and 学生表.院系名称 in(Select distinct 院系名称 from 学生表)
```

3。 带 some | any | all 的嵌套查询

语法：

```
scalar_expression{=|<>|!=|>|>=|!>|<|<=|!<}
{some|any|ALL}(子查询)
```

SOME 是 SQL 中的逻辑运算符号，如果在一系列比较中，有些为 TRUE,那么就为 TRUE。

ANY 是 SQL 中的逻辑运算符号,如果在一系列比较中,任何一个为 TRUE,那么就为 TRUE。

ALL 是 SQL 中的逻辑运算符号，如果在一系列比较中，全部都为 TRUE,那么就为 TRUE。

带 any 的嵌套查询和 some 的嵌套查询功能是一样的。

早期的 SQL 仅仅允许使用 any，后来的版本为了和英语的 any 相区分，引入了 some，同时还保留了 any 关键词。

```
select degree from sclass where cno='101'
go
select *
from sclass
where cno='101' and degree >some|any(select degree from sclass where cno='101')
go
```

```
select sno from sclass where cno='101'
go
select *
from student
where sno=some|any(select sno from sclass where cno='101')
go
select 学生表.学号,学生表.姓名,学生表.院系名称,成绩表.课程代号,成绩表.课程成绩
from 成绩表,学生表
where 成绩表.学号=学生表.学号
and 学生表.院系名称 =some|any(select 院系名称 from 学生表 where 性别='女')
select 学生表.学号,学生表.姓名,学生表.院系名称,成绩表.课程代号,成绩表.课程成绩
from 成绩表,学生表
where 成绩表.学号=学生表.学号
and 成绩表.课程成绩 <all(select 课程成绩 from 成绩表 where 课程代号
='2002030001')

select emp.empno,emp.ename,emp.job,emp.sal
from scott.emp
where sal >some|any(select sal from scott.emp where job='MANAGER');
```

带 any 的查询过程等价于两步的执行过程。

(1)执行“select sal from scott.emp where job='MANAGER'”,其结果如图 4.22 所示。

(2)查询到 3 个薪水值 2975、2850 和 2450，父查询执行下列语句。

```
   select emp.empno,emp.ename,emp.job,emp.sal from scott.emp
where sal >2975 or sal>2850 or sal>2450;
   SELECT emp.empno,emp.ename,emp.job,emp.sal
FROM scott.emp
where sal >all(select sal from scott.emp where job='MANAGER');
```

带 all 的嵌套查询与【some】的步骤相同。

(1)子查询，结果如图 4.22 所示。

(2)父查询执行下列语句。

```
SELECT emp.empno,emp.ename,emp.job,emp.sal
FROM scott.emp
WHERE sal >2975 and sal>2850 and sal>2450;
----------------------------
```

4。 带 exists 的嵌套查询

EXISTS 是 SQL 中的逻辑运算符号，如果子查询包含一些行，那么就为 TRUE。

语法为：exists 子查询

子查询是一个受限的 SELECT 语句，不允许有 COMPUTE 子句和 INTO 关键字。

exists 为存在之意，它只查找满足条件的哪些记录，一旦找到第一个匹配的记录后，就马上停

止查找
。
```
SELECT column1 FROM table1 WHERE EXISTS ( SELECT column1 FROM table2 WHERE
table1.column1
= table2.column1 );
select * from student
where exists(select * from sclass where sclass.sno=student.sno)
select * from 学生表
where exists(select * from 成绩表 where 成绩表.学号=学生表.学号)

    SELECT emp.empno,emp.ename,emp.job,emp.sal
FROM scott.emp,scott.dept
WHERE exists (SELECT * FROM scott.emp WHERE
scott.emp.deptno=scott.dept.deptno);
----------------------------
```
5。交并差操作：
交操作的嵌套查询：
交操作就是集合中交集的概念。属于集合 A 且属于集合 B 的元素总和就是交集。
```
(select deptno from scott.emp) intersect (select deptno from
scott.dept);
```
并操作的嵌套查询：
并操作就是集合中并集的概念。属于集合 A 或集合 B 的元素总和就是并集。
```
(select id,sno from student) union (select id,sno from sclass);
(select deptno from scott.emp) union (select deptno from
scott.dept);
```
差操作的嵌套查询：
差操作就是集合中差集的概念。属于集合 A 且不属于集合 B 的元素总和就是差集。
```
(select deptno from scott.dept) minus (select deptno from scott.emp);
```
《并、交和差操作的嵌套查询要求属性具有相同的定义，包括类型和取值范围。》
```
----------------------------
```
总结：
性能：
子查询很重要的一个方面就是性能表现。便利性是有代价的，
它取决于你所使用的表和声明的大小，数量和复杂性，还有你可能会允许你的
应用软件做处理工作。每一个查询在被主查询作为资源使用之前，
都将被完整地单独处理。如果可能的话，创造性地使用 JOIN 声明可以以较少的
滞后时间提供出相同的信息。
技巧：
子查询除非能确保内层 select 只返回一个行的值，否则应在外层 where 子句中用
一个 in 限定符，即要返回多个值，要用 in 或者 not in 哦，所以当在编译过程中出

现“子查询只返回一个值”的错误时，就要考虑是不是要用 in 和 not in
罗一恒,中学物理教师,除了热爱教育行业,业余时间自学 ASP.NET 技术,让技术实现梦想!志同道合的朋友请加 QQ:1006

变量直接赋值语句

【变量直接赋值的写法】

```
DECLARE @SQL VARCHAR(MAX);
select @SQL= stuff((select ''','''+DM FROM V_VIPSET for xml path('')),1,2,'')
SELECT @SQL
```

【对字符串操作如下图处理方案】

```
DECLARE @SQL VARCHAR(max);
select @SQL= stuff((select ''','''+DM FROM V_VIPSET for xml path('')),1,2,'')
SELECT @SQL=@SQL+''''
SELECT @SQL
```

不用 Pivot,Group BY，游标来进行行转列

```
------------------子查询的使用
SELECT DISTINCT T.姓名,
(select max(成绩) FROM CTE where 姓名=t.姓名 AND 科目 = '语文') AS '语文',
(select max(成绩) FROM CTE where 姓名=t.姓名 AND 科目 = '数学') AS '数学',
----------------------------【经典的子查询的使用，判断其使用的意思】
(select max(成绩) FROM CTE where 姓名=t.姓名 AND 科目 = '英语') AS '英语'
--max(CASE CTE.科目 WHEN '语文' THEN CTE.成绩 ELSE 0 END) AS 语文,
--max(CASE CTE.科目 WHEN '数学' THEN CTE.成绩 ELSE 0 END) AS 数学,
--max(CASE CTE.科目 WHEN '英语' THEN CTE.成绩 ELSE 0 END) AS 英语
 FROM CTE T
 --GROUP BY CTE.姓名
```

NOT Exists 的使用熟悉

往@Table1 中插入数据，如果存在的话就不用插入。。使用条件来判断。

```
SELECT * FROM @TABLE1 A
```

WHERE NOT EXISTS (SELECT 1 FROM @TABLE2)
-------重点的使用说明 【Table2 表中存在数据的话，就不查询，，如果 Table2 中没有数据的话，就查询出 Table1 的数据。以 Table2 为基准来判断是否正确插入数据。】

update bak_qdtjdmh set New_mxbh=(select mxbh from qdtjdmh where old_mxbh=bak_qdtjdmh.mxbh) 【更新从其他表中的数据更新过来】

游标的学习,主要用来处理判断语句的

```
/*
   建议编制计划任务，在数据库空闲时执行，否则收缩可能会失败
*/
use tempdb
go
--清除数据库中由BSERP程序创建的临时表
declare @strqry varchar(1000)
declare @temptable varchar(200)
declare @temtype varchar(200)

DECLARE Droptemptable CURSOR FOR
SELECT name,xtype FROM dbo.sysobjects where name like '%BSTEMP%' or name like '%SPCBB%'     【游标的申明】

OPEN Droptemptable

FETCH NEXT FROM Droptemptable INTO @temptable,@temtype
【打开游标，进行遍历测试】


WHILE @@FETCH_STATUS = 0
BEGIN
  if @temtype='V'
  begin
     SET @strqry=' DROP VIEW '+@temptable
     exec(@strqry)
  end
  if @temtype='U'
```

```
  begin
     SET @strqry=' DROP TABLE '+@temptable
     exec(@strqry)
  end
```

【循环判断游标里面的语句。】

```
  FETCH NEXT FROM Droptemptable   INTO @temptable,@temtype
END


CLOSE Droptemptable
```

【关闭游标】

```
DEALLOCATE Droptemptable
```

【删除游标引用。当释放最后的游标引用时，组成该游标的数据结构由 Microsoft® SQL Server™释放。】

```
go

--收缩数据文件
dbcc shrinkfile (tempdev, 0)
go

--收缩日志文件
dbcc shrinkfile (templog, 0)
go
```

触发器 deleted 表和 inserted 表详解!!!

```
create trigger updateDeleteTime
on user
for update
as
begin
 update user set UpdateTime=(getdate()) from user inner join inserted on use
r.UID=Inserted.UID
end
```

上面的例子是在执行更新操作的时候同时更新，一下修改时间。

关键在于 Inserted 表

触发器语句中使用了两种特殊的表：**deleted** 表和 **inserted** 表。

数据库项目文档

Deleted 表用于存储 DELETE 和 UPDATE 语句所影响的行的复本。在执行 DELETE 或 UPDATE 语句时，行从触发器表中删除，并传输到 deleted 表中。Deleted 表和触发器表通常没有相同的行。
Inserted 表用于存储 INSERT 和 UPDATE 语句所影响的行的副本。在一个插入或更新事务处理中，新建行被同时添加到 inserted 表和触发器表中。Inserted 表中的行是触发器表中新行的副本。
1.插入操作（Insert）
Inserted 表有数据，Deleted 表无数据

2.删除操作（Delete）
Inserted 表无数据，Deleted 表有数据

3.更新操作（Update）
Inserted 表有数据（新数据），Deleted 表有数据（旧数据）
应用实例

⊞

⊟
代码

```
set ANSI_NULLS ON
set QUOTED_IDENTIFIER ON
go


-- =============================================
-- Author:        <Author,sufei>
-- Create date: <Create Date,2010-05-11>
-- Description:    <当是短信充值时修改相信的记录使记录不会重复获取>
-- =============================================
ALTER TRIGGER [dbo].[updatestart]
   ON [dbo].[OrderTelecom] FOR update
AS
BEGIN

    DECLARE @state int;
    DECLARE @note2 varchar(50)

    SELECT  @state= Inserted.ortState,@note2 =Inserted.ortNote2 from Inserte
d

    IF @state=1 AND @note2=1
     begin
```

```
        --当发短信猫取走记录时修改状态为成功和取过的状态
       update OrderTelecom set OrderTelecom.ortState=2 ,OrderTelecom.ortSmsme
ssages='短信充值成功'
       from OrderTelecom inner join Inserted on OrderTelecom.ortId=Inserted.o
rtId
      end

      if @state in(2,3,10) and @note2=0
       begin
       update OrderTelecom set ortNote2=1
       from OrderTelecom inner join Inserted on OrderTelecom.ortId=Inserted.o
rtId
      end

END
```

触发器的实战使用

```
/****** 对象:  Trigger [qdtjd_insert]    脚本日期: 11/17/2011 12:23:18
******/
IF  EXISTS (SELECT * FROM sys.triggers WHERE object_id =
OBJECT_ID(N'[dbo].[qdtjd_insert]'))
DROP TRIGGER [dbo].[qdtjd_insert]
GO
CREATE TRIGGER [qdtjd_insert]
   ON  [dbo].[QDTJD]
  FOR insert                                          【Insert触发器】
AS
Begin
insert into bak_qdtjd select *,'0',getdate() from inserted
【某张表插入时候的日期，使用触发器后,Inserted,Deleted就相当于一张表了】
End
GO

/****** 对象:  Trigger [qdtjd_update]    脚本日期: 11/17/2011 12:23:49
******/
IF  EXISTS (SELECT * FROM sys.triggers WHERE object_id =
OBJECT_ID(N'[dbo].[qdtjd_update]'))
DROP TRIGGER [dbo].[qdtjd_update]
GO
```

```
CREATE TRIGGER [qdtjd_update]
   ON  [dbo].[QDTJD]
  FOR update
AS
Begin
update bak_qdtjd set rq=(select rq from inserted),ydjh=(select ydjh from
inserted),
cj=(select cj from inserted),jz=(select jz from inserted),rq_1=(select
rq_1 from inserted),
rq_2=(select rq_2 from inserted),bz=(select bz+'('+DJBH+')' from
inserted),jzrq=(select jzrq from inserted),
sp=(select sp from inserted),sprq=(select sprq from inserted) where
djbh=(select djbh from inserted)
End
GO

/****** 对象:  Trigger [qdtjd_delete]    脚本日期: 11/17/2011 12:22:46
******/
IF  EXISTS (SELECT * FROM sys.triggers WHERE object_id =
OBJECT_ID(N'[dbo].[qdtjd_delete]'))
DROP TRIGGER [dbo].[qdtjd_delete]
GO
CREATE TRIGGER [qdtjd_delete]
   ON  [dbo].[QDTJD]
  FOR delete
AS
Begin
UPDATE bak_qdtjd SET DELETED=1 WHERE djbh in (SELECT djbh FROM DELETED)
End
Go


--停用相关触发器,对相关的数据库进行相触
ALTER TABLE qdtjdmp DISABLE TRIGGER qdtjdmp_insert
ALTER TABLE qdtjdmh DISABLE TRIGGER qdtjdmh_insert
```

快速查询一张表的数据,查询一个数据库中有多少表

```
SELECT count(1) from sysobjects where xtype = 'u'
sp_spaceused 'Fact_MonthlyStock'  【函数的讲解】
```

赋值方式的学习中

```
--变量的赋值方式学习中。请注意思考。。从不同角度入手进行结算的分析。
DECLARE @IndipendentAccount INT;
SET @IndipendentAccount = 0;
SELECT
    @IndipendentAccount = FLAG10
FROM MAIN2
WHERE ID = 5;
```

不同其他的左外连接使用

```
  LEFT JOIN PINPAI
            LEFT JOIN yangpin
              ON PINPAI.PPDM  = yangpin.PPDM
            LEFT JOIN DALEI
              ON YANGPIN.BYZD4 = DALEI.DLDM
            LEFT JOIN JIJIE
              ON YANGPIN.BYZD5 = JIJIE.JJDM
       ON DHJRDMX.SPDM = YANGPIN.YPDM
```

SSRS 学习子项占总项的比

子项总总项的比，求法如下：

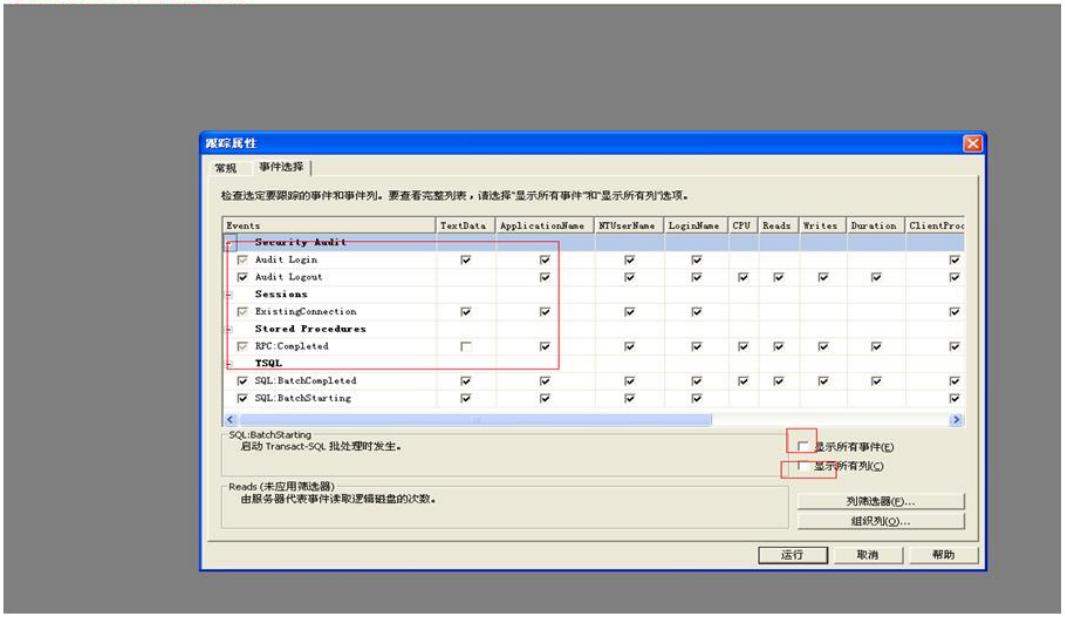
然后我们填入具体的表达式：

=SUM(Fields!LineTotal.Value)/SUM(Fields!LineTotal.Value,"ProductCategoryName")

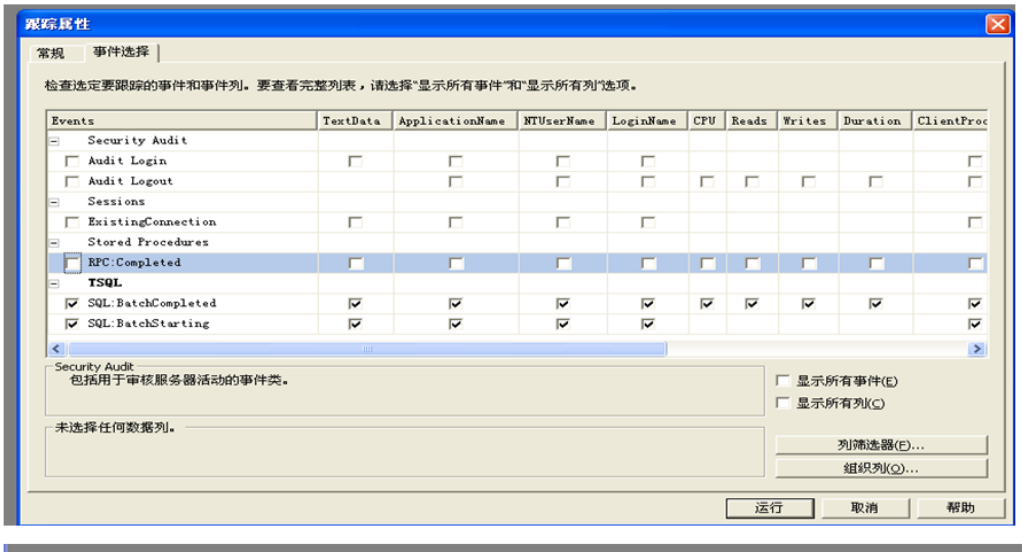
数据库项目文档

SQL SERVER Profiler 的使用，只要自己机器上的语句

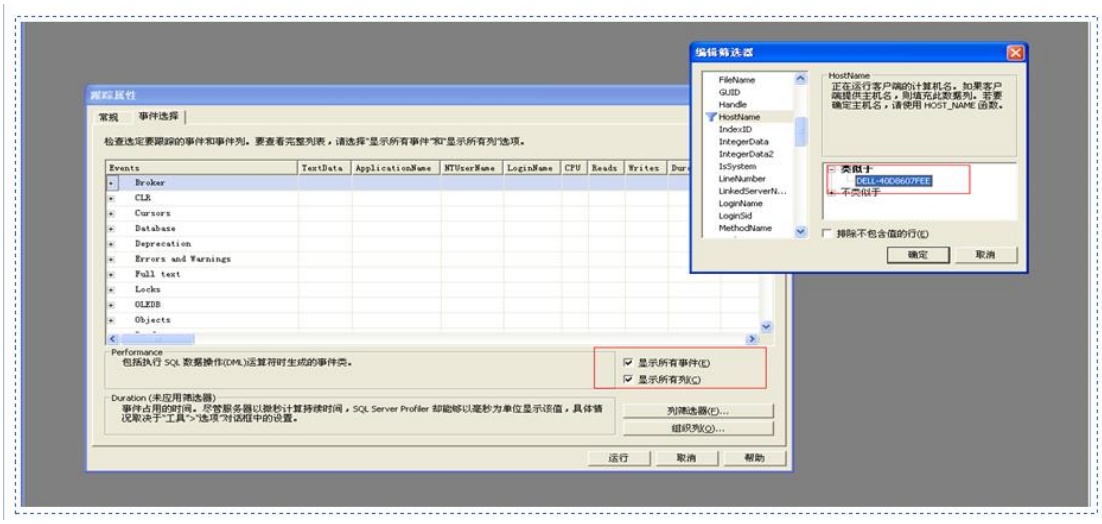
1.选择显示所有事件，和显示所有列。

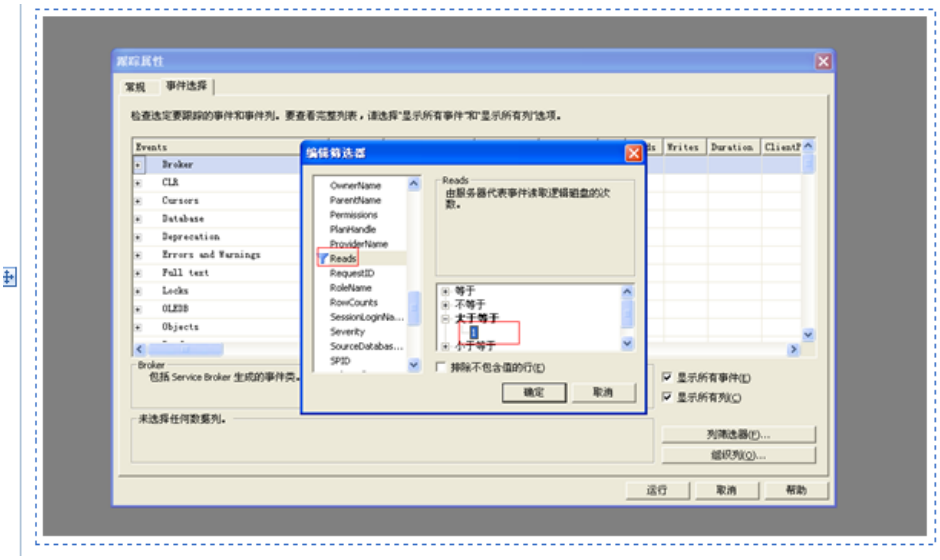


2.设置后的如下.勾选 显示所有事件，和显示所有列，并设置HostName 为自己的机器 DELL-40D8607FEE.Reads设置为>=1,不然会出来很多没有用到的信息的.

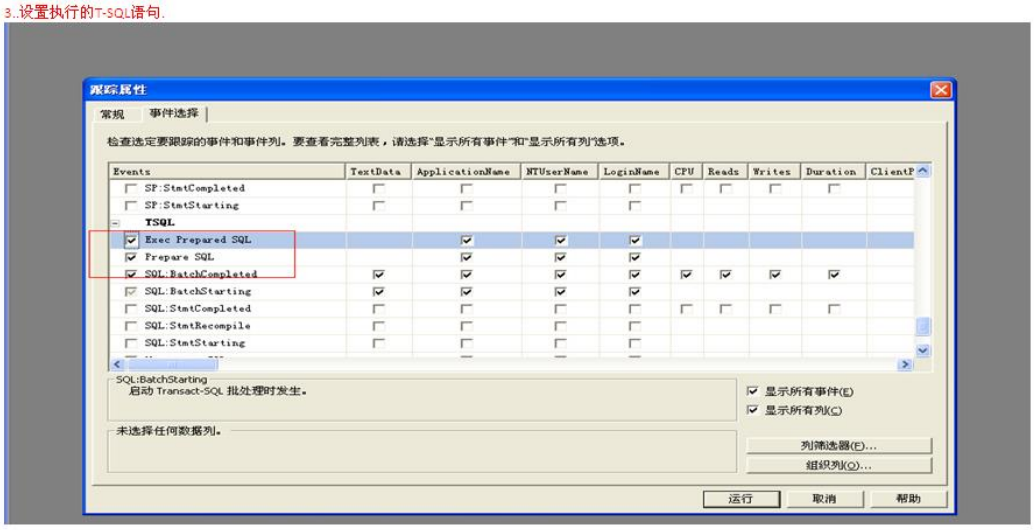


数据库项目文档

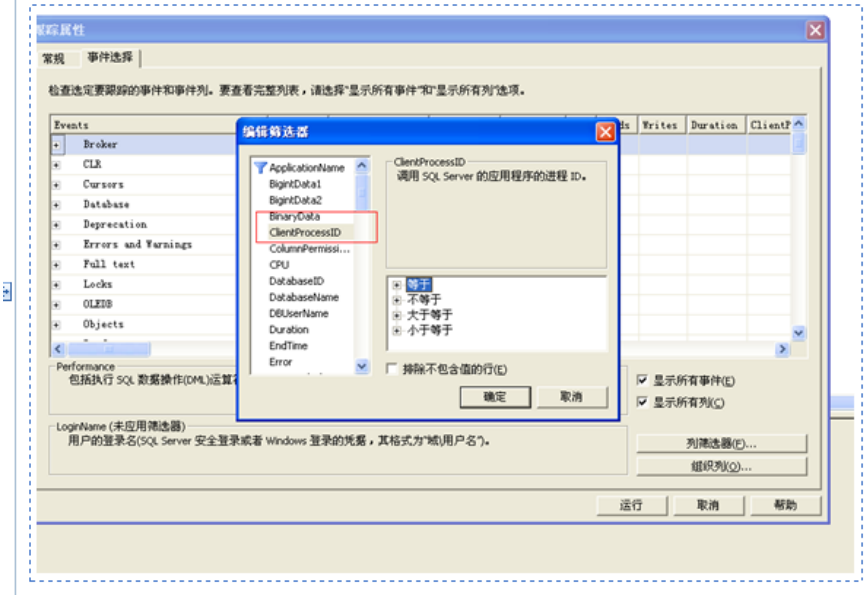




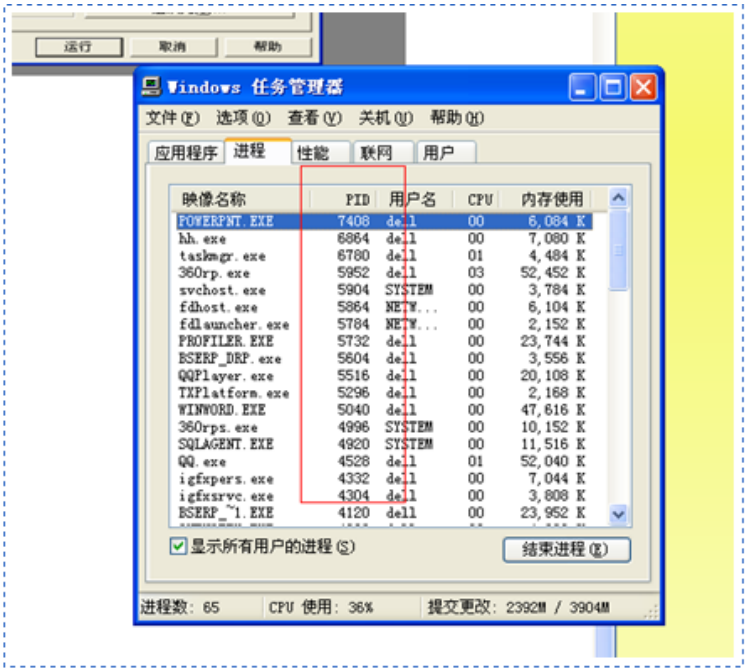
3..设置执行的T-SQL语句.



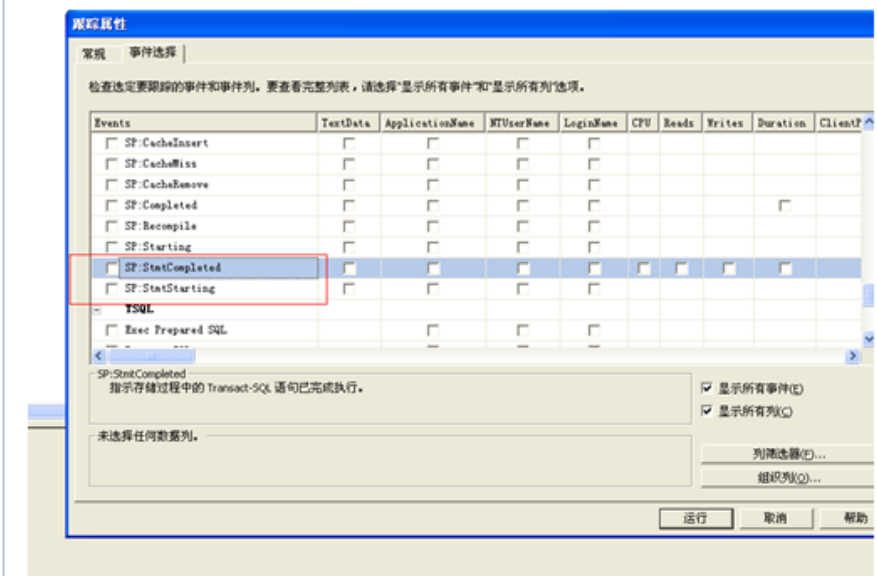
数据库项目文档



数据库项目文档



对存储过程子语句的执行那个，存储过程报错，都可以跟踪使用



用语句查找视图，知道视图的名称就可以了

sp_helptext VT_QTLSD　显示的文本

显示用户定义规则的定义、默认值、未加密的 Transact-SQL 存储过程、用户定义 Transact-SQL 函数、触发器、计算列、CHECK 约束、视图或系统对象（如系统存储过程）。

数据库项目文档

源文档 <http://msdn.microsoft.com/zh-cn/library/ms176112.aspx>

查找数据库中所有的视图

```
select views.Name as [视图名],Col.Name as [列名] ,Type.name as [数据类
型],Col.max_length as [字段长度]
        --,Col_Desc.Value as Col_Description
from sys.views views
inner join sys.columns Col on views.object_id  = Col.object_id
inner join sys.types Type on Col.system_type_id = Type.system_type_id
--left join sys.extended_properties Col_Desc
--    on Col_Desc.major_id=views.object_id and Col_Desc.minor_id=Col.Column_id and
Col_Desc.class=1
order by Create_Date
```

查找数据库中所有的存储过程

```
select Pr_Name as [存储过程], [参数]=stuff((select ', '+[Parameter]
from (
select Pr.Name as Pr_Name,parameter.name +' ' +Type.Name + '
('+convert(varchar(32),parameter.max_length)+')' as Parameter
from sys.procedures Pr left join
sys.parameters parameter on Pr.object_id = parameter.object_id
inner join sys.types Type on parameter.system_type_id =
Type.system_type_id
where type = 'P'
) t where Pr_Name=tb.Pr_Name for xml path('')), 1, 1, '')
from (
select Pr.Name as Pr_Name,parameter.name +' ' +Type.Name + '
('+convert(varchar(32),parameter.max_length)+')' as Parameter
from sys.procedures Pr left join
sys.parameters parameter on Pr.object_id = parameter.object_id
inner join sys.types Type on parameter.system_type_id =
Type.system_type_id
where type = 'P'
)tb
where Pr_Name not like 'sp_%' --and Pr_Name not like 'dt%'
group by Pr_Name
order by Pr_Name
```

天善日期类型格式

```
CONVERT(varchar(8), DINGHUOHUI.RQ, 112)     天善日期格式
```

T-SQL 中 GOTO 学习使用

GOTO 的语法学习如下，请记录:

```
RIT:
  COMMIT TRAN;
  SELECT 0 AS ErrorCount;
  RETURN;
ERR:
  ROLLBACK TRAN;
  SELECT 1 AS ErrorCount;
  RETURN;
```

清除 SQL 查询缓存

```
-----清除缓存，每次的查询效率都一样的做法。。。
dbcc dropcleanbuffers
```

SQL 获取本月最后一天

```
select convert(char(10),dateadd(d,-1,convert(char(7),
dateadd(m,1,getdate()),120)+'-01'),120)     -----获取本月最后一天


convert(varchar(8),QRRQ,112)    【天善日期转化形式】

  select @MinRQ=max(RQ) from VW_CKJXCMX AS A
  select
@MinMonth=convert(varchar(8),DATEADD(Month,1,convert(datetime,convert
(varchar(6),@MinRQ,112)+'01'))-1,112);  -- 获取某一天得最后一天
  select
@MaxMonth=convert(varchar(8),convert(datetime,convert(varchar(6),getd
ate(),112)+'01')-1,112);   --获取上月最后一天
```

```
convert(varchar(8),DateAdd(MM,-3,GetDate()),112)
```
前三个月的数据.

```
convert(varchar(8),sg_gathering.dtDate,114) as IntervalCode,
```
转化为小时的做法

硬件方面的学习

CPU(中央处理器):

※Intel 桌面：赛扬、奔腾、酷睿 2 、酷睿 i3、酷睿 i5、酷睿 i7
※Intel 移动：凌动、赛扬、奔腾、酷睿 2、酷睿 i3、酷睿 i5、酷睿 i7
※AMD 桌面：闪龙、速龙、羿龙、速龙 II、羿龙 II
※AMD 移动：锐龙、闪龙、速龙、速龙 II、羿龙 II

本机 Inter(R) Core(TM) i7 CPU M 640 @2.80GHZ

内存(存储器)：存储器是用来存储程序和数据的部件，对于计算机来说，有了存储器，才有记忆功能，才能保证正常工作。存储器的种类很多，按其用途可分为主存储器和辅助存储器，主存储器又称内存储器（简称内存，港台称之为记忆体）。当然内存的好坏会直接影响电脑的运行速度。内存就是存储程序以及数据的地方，比如当我们在使用 WPS 处理文稿时，当你在键盘上敲入字符时，它就被存入内存中，当你选择存盘时，内存中的数据才会被存入硬（磁）盘。 内存一般采用半导体存储单元，包括随机存储器（RAM），只读存储器（ROM），以及高速缓存（CACHE）。

内存大小： 如果你的系统拥有 4GB 内存或 RAM，还有一块 512MB 显存的显卡，那么只有其中的 3GB 左右是可以得到充分利用的。如果想要换块更大显存的显卡，或者想要用 2 块显卡进行混合交火或 SLI 双路渲染的话，最好还是考虑一下换用 64 位 Vista。

显卡：显卡的用途是将计算机系统所需要的显示信息进行转换驱动，并向显示器提供行扫描信号，控制显示器的正确显示，是连接显示器和个人电脑主板的重要元件，是“人机对话”的重要设备之一。显卡作为电脑主机里的一个重要组成部分，承担输出显示图形的任务，对于从事专业图形设计的人来说显卡非常重要。 民用显卡图形芯片供应商主要包括 AMD（ATI）和 Nvidia(英伟达)两家。
数据（data） 一旦离开 CPU，必须通过 4 个步骤，最后才会到达显示屏：

硬盘：它使磁头灵敏度进一步提升，进而提高了储存密度 。其桌面产品分为侧重高 IO 性能的 Black 系列（俗称“黑盘”），普通的 Blue 系列（俗称蓝盘），以及侧重低功耗、低噪音的环保 Green 系

列（俗称绿盘）。 磁头是硬盘中最昂贵的部件，也是硬盘技术中最重要和最关键的一环。传统的磁头是读写合一的电磁感应式磁头，但是，硬盘的读、写却是两种截然不同的操作，为此，这种二合一磁头在设计时必须要同时兼顾到读/写两种特性，从而造成了硬盘设计上的局限。而 MR 磁头（Magnetoresistive heads），即磁阻磁头，采用的是分离式的磁头结构：写入磁头仍采用传统的磁感应磁头（MR 磁头不能进行写操作），读取磁头则采用新型的 MR 磁头，即所谓的感应写、磁阻读。这样，在设计时就可以针对两者的不同特性分别进行优化，以得到最好的读/写性能。另外，MR 磁头是通过阻值变化而不是电流变化去感应信号幅度，因而对信号变化相当敏感，读取数据的准确性也相应提高。而且由于读取的信号幅度与磁道宽度无关，故磁道可以做得很窄，从而提高了盘片密度，达到 200MB/英寸 2，而使用传统的磁头只能达到 20MB/英寸 2，这也是 MR 磁头被广泛应用的最主要原因。目前，MR 磁头已得到广泛应用，而采用多层结构和磁阻效应更好的材料制作的 GMR 磁头（Giant Magnetoresistive heads）也逐渐普及。

转速(Rotationl Speed 或 Spindle speed)，是硬盘内电机主轴的旋转速度，也就是硬盘盘片在一分钟内所能完成的最大转数。转速的快慢是标示硬盘档次的重要参数之一，它是决定硬盘内部传输率的关键因素之一，在很大程度上直接影响到硬盘的速度。硬盘的转速越快，硬盘寻找文件的速度也就越快，相对的硬盘的传输速度也就得到了提高。硬盘转速以每分钟多少转来表示，单位表示为 RPM，RPM 是 Revolutions Per minute 的缩写，是转/每分钟。RPM 值越大，内部传输率就越快，访问时间就越短，硬盘的整体性能也就越好。

硬盘的主轴马达带动盘片高速旋转，产生浮力使磁头飘浮在盘片上方。要将所要存取资料的扇区带到磁头下方，转速越快，则等待时间也就越短。因此转速在很大程度上决定了硬盘的速度。

磁盘阵列

磁盘阵列（Redundant Arrays of Inexpensive Disks，RAID），有“价格便宜且多余的磁盘阵列”之意。原理是利用数组方式来作磁盘组，配合数据分散排列的设计，提升数据的安全性。磁盘阵列是由很多便宜、容量较小、稳定性较高、速度较慢磁盘，组合成一个大型的磁盘组，利用个别磁盘提供数据所产生加成效果提升整个磁盘系统效能。同时利用这项技术，将数据切割成许多区段，分别存放在各个硬盘上。磁盘阵列还能利用同位检查（Parity Check）的观念，在数组中任一颗硬盘故障时，仍可读出数据，在数据重构时，将数据经计算后重新置入新硬盘中。

RAID 技术主要包含 RAID 0～RAID 7 等数个规范，它们的侧重点各不相同，常见的规范有如下几种： RAID 0：RAID 0 连续以位或字节为单位分割数据，并行读/写于多个磁盘上，因此具有很高的数据传输率，但它没有数据冗余，因此并不能算是真正的 RAID 结构。RAID 0 只是单纯地提高性能，并没有为数据的可靠性提供保证，而且其中的一个磁盘失效将影响到所有数据。因此，RAID 0 不能应用于数据安全性要求高的场合。 RAID 1：它是通过磁盘数据镜像实现数据冗余，在成对的独立磁盘上产生互为备份的数据。当原始数据繁忙时，可直接从镜像拷贝中读取数据，因此 RAID 1 可以提高读取性能。RAID 1 是磁盘阵列中单位成本最高的，但提供了很高的数据安全性和可用性。当一个磁盘失效时，系统可以自动切换到镜像磁盘上读写，而不需要重组失效的数据。 RAID 0+1：也被称为 RAID 10 标准，实际是将 RAID 0 和 RAID 1 标准结合的产物，在连续地以位或字节为单位分割数据并且并行读/写多个磁盘的同时，为每一块磁盘作磁盘镜像进行冗余。它的优点是同时拥有 RAID 0 的超凡速度和 RAID 1 的数据高可靠性，但是 CPU 占用率同样也更高，而且磁盘的利用率比较低。 RAID 2：将数据条块化地分布于不同的硬盘上，条块单位为位或字节，并使用称为“加重平均纠错码（海明码）”的编码技术来提供错误检查及恢复。这种编码技术需要多个磁盘存放检查及恢复信息，使得 RAID 2 技术实施更复

杂，因此在商业环境中很少使用。　　RAID 3：它同 RAID 2 非常类似，都是将数据条块化分布于不同的硬盘上，区别在于 RAID 3 使用简单的奇偶校验，并用单块磁盘存放奇偶校验信息。如果一块磁盘失效，奇偶盘及其他数据盘可以重新产生数据；如果奇偶盘失效则不影响数据使用。RAID 3 对于大量的连续数据可提供很好的传输率，但对于随机数据来说，奇偶盘会成为写操作的瓶颈。　　RAID 4：RAID 4 同样也将数据条块化并分布于不同的磁盘上，但条块单位为块或记录。RAID 4 使用一块磁盘作为奇偶校验盘，每次写操作都需要访问奇偶盘，这时奇偶校验盘会成为写操作的瓶颈，因此 RAID 4 在商业环境中也很少使用。　　RAID 5：RAID 5 不单独指定的奇偶盘，而是在所有磁盘上交叉地存取数据及奇偶校验信息。在 RAID 5 上，读/写指针可同时对阵列设备进行操作，提供了更高的数据流量。RAID 5 更适合于小数据块和随机读写的数据。RAID 3 与 RAID 5 相比，最主要的区别在于 RAID 3 每进行一次数据传输就需涉及到所有的阵列盘；而对于 RAID 5 来说，大部分数据传输只对一块磁盘操作，并可进行并行操作。在 RAID 5 中有“写损失”，即每一次写操作将产生四个实际的读/写操作，其中两次读旧的数据及奇偶信息，两次写新的数据及奇偶信息。　　RAID 6：与 RAID 5 相比，RAID 6 增加了第二个独立的奇偶校验信息块。两个独立的奇偶系统使用不同的算法，数据的可靠性非常高，即使两块磁盘同时失效也不会影响数据的使用。但 RAID 6 需要分配给奇偶校验信息更大的磁盘空间，相对于 RAID 5 有更大的“写损失”，因此“写性能”非常差。较差的性能和复杂的实施方式使得 RAID 6 很少得到实际应用。　　RAID 7：这是一种新的 RAID 标准，其自身带有智能化实时操作系统和用于存储管理的软件工具，可完全独立于主机运行，不占用主机 CPU 资源。RAID 7 可以看作是一种存储计算机（Storage Computer），它与其他 RAID 标准有明显区别。除了以上的各种标准（如表 1），我们可以如 RAID 0+1 那样结合多种 RAID 规范来构筑所需的 RAID 阵列，例如 RAID 5+3（RAID 53）就是一种应用较为广泛的阵列形式。用户一般可以通过灵活配置磁盘阵列来获得更加符合其要求的磁盘存储系统。

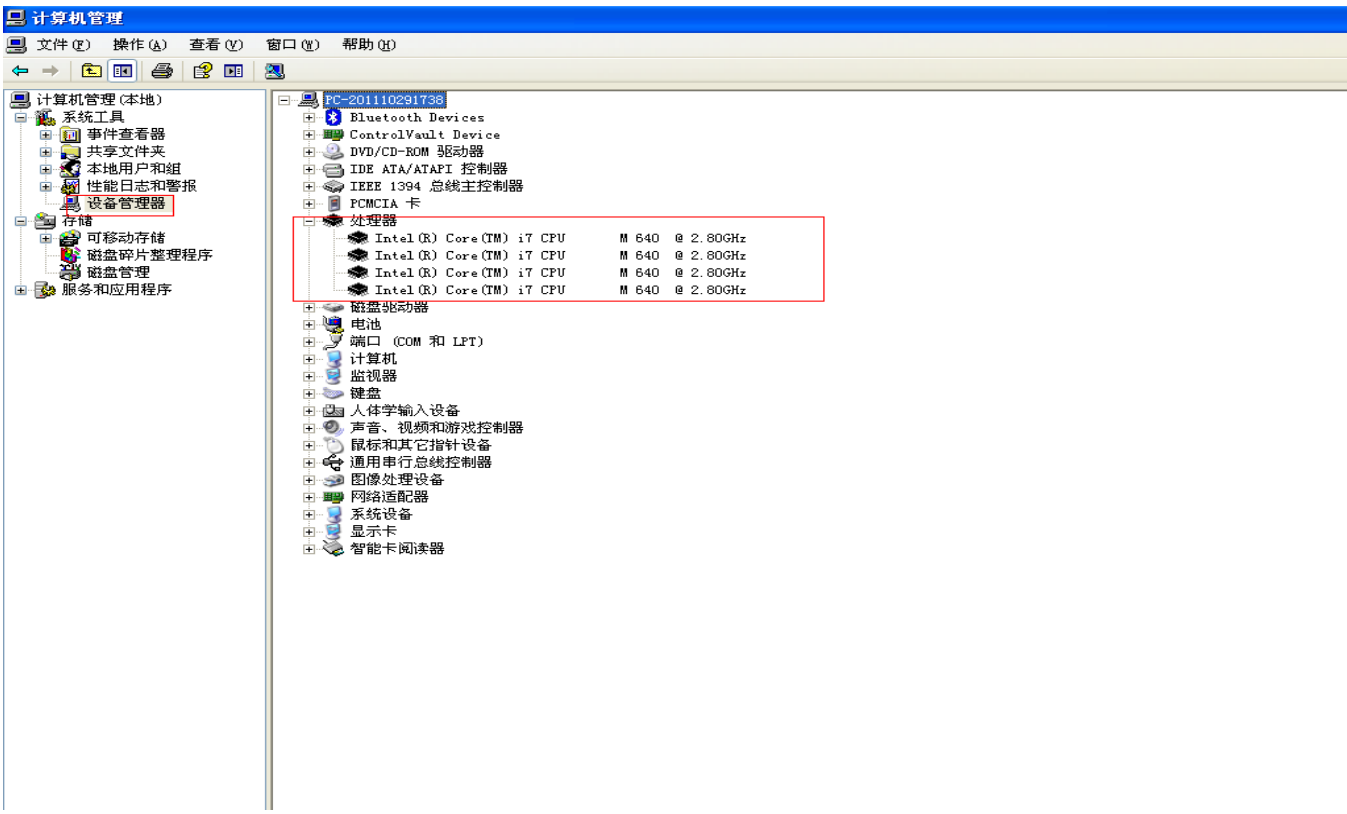
简单的说，RAID 是一种把多块独立的硬盘（物理硬盘）按不同的方式组合起来形成一个硬盘组（逻辑硬盘），从而提供比单个硬盘更高的存储性能和提供数据备份技术。组成磁盘阵列的不同方式成为 RAID 级别（RAID Levels）。数据备份的功能是在用户数据一旦发生损坏后，利用备份信息可以使损坏数据得以恢复，从而保障了用户数据的安全性。在用户看起来，组成的磁盘组就像是一个硬盘，用户可以对它进行分区，格式化等等。总之，对磁盘阵列的操作与单个硬盘一模一样。不同的是，磁盘阵列的存储速度要比单个硬盘高很多，而且可以提供自动数据备份。

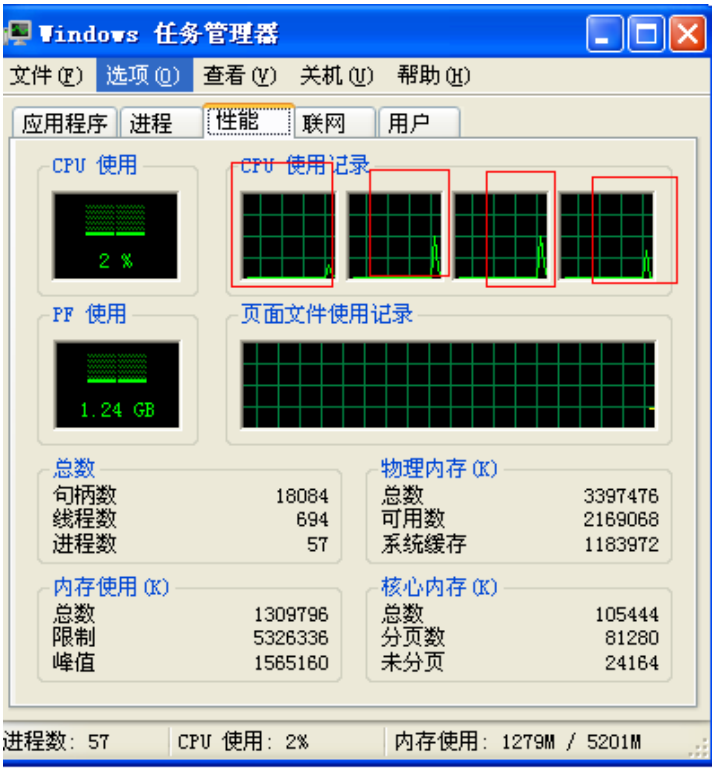
IO

I/O 输出/输入(Input/Output)，分为 IO 设备和 IO 接口两个部分。　　在 POSIX 兼容的系统上，例如 Linux 系统，I/O 操作可以有多种方式，比如 DIO(Direct I/O)，AIO(Asynchronous I/O 异步 I/O)，Memory-Mapped I/O(内存映设 I/O)等，不同的 I/O 方式有不同的实现方式和性能，在不同的应用中可以按情况选择不同的 I/O 方式。　　输入输出 I/O 流可以近似的看成对字节或者包装后的字节的读取就是拿出来放进去双路切换；实现联动控制系统的弱电线路与被控设备的强电线路之间的转接、隔离，以防止强电窜入系统，保障系统的安全；　　与专线控制盘连接，用于控制重要消防设备（如消防泵、喷淋泵、风机等），一只模块可控制一台大型消防设备的启、停控制；

几核 CPU：在计算机管理里面可以看到，在任务管理器里面也可看到。

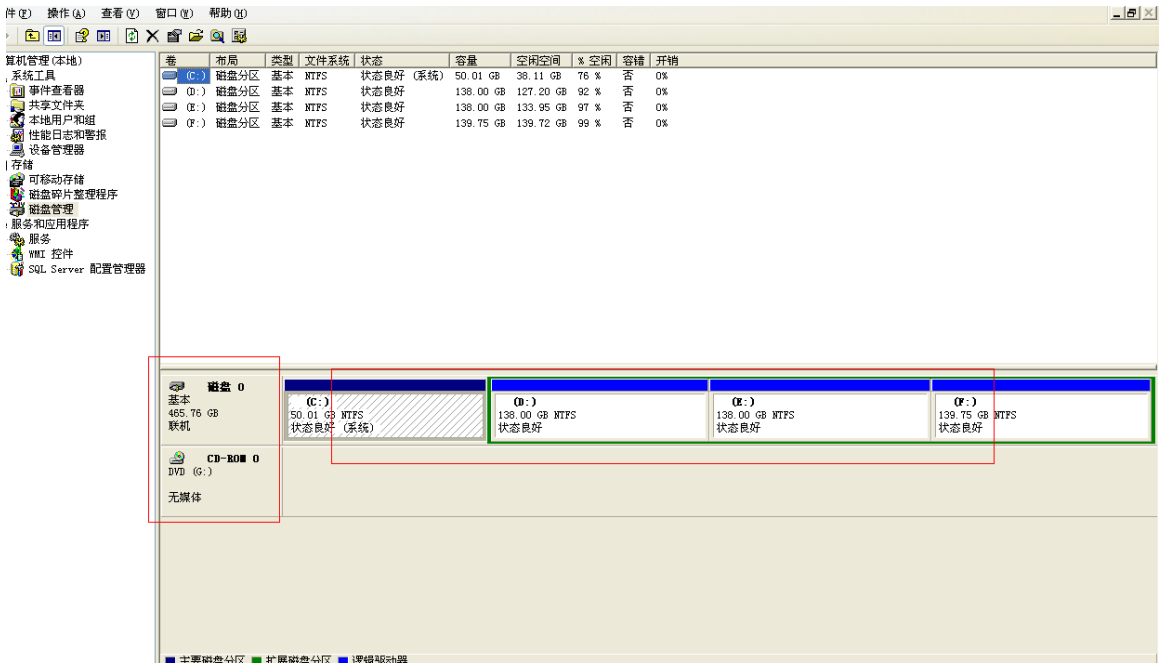
数据库项目文档





数据库项目文档

几块磁盘：有几块磁盘，每个磁盘有几块分区在这里都可以看到



字符串截取，截取两边的，截取中间的等等

```
DeCLARE @STR VARCHAR(500),@SQL VARCHAR(500);
SET @STR = 'LYTES<<8888>>中';
SET @SQL='1234,5678';
SELECT
substring(@STR,charindex('<<',@str)+2,charindex('>>',@str)-charindex(
'<<',@str)-2)

SELECT LEFT(@SQL,CHARINDEX(',',@SQL)-1)
SELECT RIGHT(@SQL,CHARINDEX(',',@SQL)-1)
```

SQL 递归的使用,依次获取上级节点并进行相应的判断

```
DECLARE @tableA TABLE(AID VARCHAR(50),[TYPE] INT);
DECLARE @tableB TABLE(BID VARCHAR(50),ShangJiB VARCHAR(50),Number INT);
INSERT INTO @tableA
SELECT '000',1 UNION ALL
SELECT 'AAA001',1 UNION ALL
SELECT 'AAABBB001',1 UNION ALL
```

```
SELECT 'XXXXX001',2 UNION ALL
SELECT 'XXXXXYYYYY001',1

INSERT INTO @tableB
SELECT '000','',0 UNION ALL   --'000' 的上级不能为自身
SELECT 'AAA001','000',1 UNION ALL
SELECT 'AAABBB001','AAA001',2 UNION ALL
SELECT 'XXXXX001','AAABBB001',3 UNION ALL
SELECT 'XXXXXYYYYY001','XXXXX001',4
;with cte as(
select a.*,(case when b.type=2 then 2 else 1 end)type
from @tableB a inner join @tableA b on a.bid=b.aid
where a.shangjib=''
union all
select a.*,(case when b.type=2 or c.type=2 then 2 else 1 end)
from @tableB a inner join @tableA b on a.bid=b.aid
inner join cte c on a.shangjib=c.bid
)select * from cte
/*
BID                                             ShangJiB
Number     type
--------------------------------------------------
-------------------------------------------------- -----------
-----------
000
0          1
AAA001                                          000
1          1
AAABBB001                                        AAA001
2          1
XXXXX001                                        AAABBB001
3          2
XXXXXYYYYY001                                    XXXXX001
4          2
*/


--select BID,
--  类别=case when exists(select 1 from @tableA where AID=t.ShangJiB and
type=2) then 2 else 1 end
--from @tableB t
```

SQL 递归依次获取上级，上上级的类型代码

```
/*
    功能说明：  获取商店表
    修改说明：   Create by LY on 2011-11-2
                 Modify by LY on 2011-12-2  添加商店性质字段。如果他本身或者他的上一级或者上上级为加盟
                                                就是加盟店，否则就是直营店。直营店为，加盟店为。
*/
DECLARE @TableName VARCHAR(30)
SELECT @TableName = 'Dim_Shop';
WITH CTE_QUDAORecursion AS -------递归判断渠道的上一级，上上级渠道的类型
(
    SELECT QUDAO_Tree.QDDM,
           QUDAO_Tree.SJQD,
           CASE
             WHEN QUDAO_Tree.byzd1 > 1 THEN 1
             ELSE 0
           END AS ShopProperty
     FROM   QUDAO QUDAO_Tree
     WHERE  QUDAO_Tree.QDDM = QUDAO_Tree.SJQD
     UNION ALL
     SELECT QUDAO_Child.QDDM,
           QUDAO_Child.SJQD,
           CASE
             WHEN ( QUDAO_Child.byzd1 > 1
                    OR CTE_QUDAORecursion.ShopProperty = 1 ) THEN 1
             ELSE 0
           END AS ShopProperty
     FROM   QUDAO QUDAO_Child
           INNER JOIN CTE_QUDAORecursion
             ON QUDAO_Child.SJQD = CTE_QUDAORecursion.QDDM
     WHERE  QUDAO_Child.QDDM <> QUDAO_Child.SJQD
)
SELECT KEHU.KHDM                                                AS
ShopCode,
      KEHU.KHMC                                                 AS
ShopName,
      '商店'                                                     AS
```

```
CusProperty,
      QUDAO.QDDM                                                AS
Channel,
      QUDAO.QDMC                                                AS
ChannelName,
      KEHU.QYDM                                                 AS
AreaCode,
      KEHU.CKDM                                                 AS
StoreHouseCode,
      CONVERT(VARCHAR(8), Isnull(KHATTACH.KSRQ, '2002-01-01'), 112) AS
DebutDate,
      KHSX1.SXMC                                                AS
CusAttribute1,
      KHSX2.SXMC                                                AS
CusAttribute2,
      KHSX3.SXMC                                                AS
CusAttribute3,
      KHSX4.SXMC                                                AS
CusAttribute4,
      KHSX5.SXMC                                                AS
CusAttribute5,
      KHSX6.SXMC                                                AS
CusAttribute6,
      CTE_QUDAORecursion.ShopProperty                               AS
ShopProperty
FROM   KEHU
      LEFT JOIN KHATTACH
        ON KEHU.KHDM = KHATTACH.KHDM
      INNER JOIN QUDAO
        ON KEHU.QDDM = QUDAO.QDDM
      INNER JOIN QUYU
        ON KEHU.QYDM = QUYU.QYDM
      INNER JOIN KHSX1
        ON KEHU.KHSX1 = KHSX1.SXDM
      INNER JOIN KHSX2
        ON KEHU.KHSX2 = KHSX2.SXDM
      INNER JOIN KHSX3
        ON KEHU.KHSX3 = KHSX3.SXDM
      INNER JOIN KHSX4
        ON KEHU.KHSX4 = KHSX4.SXDM
      INNER JOIN KHSX5
```

```
        ON KEHU.KHSX5 = KHSX5.SXDM
      INNER JOIN KHSX6
        ON KEHU.KHSX6 = KHSX6.SXDM
      INNER JOIN CTE_QUDAORecursion
        ON CTE_QUDAORecursion.QDDM = QUDAO.QDDM
      --INNER JOIN FN_RDC_ChangedPK(@TableName) cdcKey
      --  ON cdcKey.PKValue = KEHU.KHDM
WHERE  KEHU.XZDM = '2'
```

列转行的终极应用

```
/*
    功能说明： 获取销售目标值
    修改说明：  使用列转行的形式。对升级到SQL Server 2005 或更高版本的数据库使用PIVOT 和UNPIVOT 时，
                必须将数据库的兼容级别设置为90 或更高。有关如何设置数据库兼容级别的信息
*/
--DECLARE @DB_Name VARCHAR(50);  --当前数据库名称
--SELECT @DB_Name = DB_NAME()
--EXEC SP_DBCMPTLEVEL @DB_Name,90
DECLARE @TableName VARCHAR(30)
SELECT @TableName = 'Fact_SaleTarget';
;WITH CTE_SaleTargetChangePK AS   ------主键改变的值
(
    SELECT CONVERT(VARCHAR(20),LEFT(PKValue,CHARINDEX(',',PKValue)-1)) AS PKValue,
          CONVERT(INT,RIGHT(PKValue,CHARINDEX(',',PKValue)-1)) AS DateKey
    FROM FN_RDC_ChangedKeyCombination(@TableName)
)
,CTE_SaleTarget AS   --------销售目标值
(
    SELECT KHDM,NFRQ,SUM(MONTH1) AS [1],SUM(MONTH2) AS [2],
         SUM(MONTH3) AS [3],SUM(MONTH4) AS [4],
         SUM(MONTH5) AS [5],SUM(MONTH6) AS [6],
         SUM(MONTH7) AS [7],SUM(MONTH8) AS [8],
         SUM(MONTH9) AS [9],SUM(MONTH10) AS [10],
         SUM(MONTH11) AS [11],SUM(MONTH12) AS [12]
```

```
    FROM SDYMBZB
    INNER JOIN CTE_SaleTargetChangePK SaleTargetChangePK ON SDYMBZB.NFRQ
= SaleTargetChangePK.DateKey
    AND SDYMBZB.KHDM = SaleTargetChangePK.PKValue
    GROUP BY KHDM,NFRQ
),
CTE_UNPIVOTSaleTarget AS
(
    SELECT KHDM,NFRQ,[Month],JE FROM (SELECT * FROM CTE_SaleTarget)P
    UNPIVOT(
    JE FOR [Month] IN
([1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12])
    )AS UNSaleTarget
)
SELECT CONVERT(INT,NFRQ+RIGHT('0'+[Month],2)+'01') AS
DateKey,CONVERT(INT,NFRQ+RIGHT('0'+[Month],2)+'01') AS ExtractDay,
      CONVERT(VARCHAR(20),KHDM) AS
CustomerCode,CONVERT(DECIMAL(18,4),0) AS
SaleQuantityPlan,CONVERT(DECIMAL(18,4),SUM(JE)) AS SaleAmountPlan
FROM CTE_UNPIVOTSaleTarget
GROUP                                                                BY
CONVERT(INT,NFRQ+RIGHT('0'+[Month],2)+'01'),CONVERT(VARCHAR(20),KHDM)
```

时间戳的使用及设计方案

```
--SELECT '1111'                AS history_db,
--      Cast(stamp AS BINARY(8)) stamp
--FROM   GRN..LSXHD
--EXCEPT
--(
-- SELECT history_db,
--        stamp stamplook
-- FROM   LSXHD
--)

/*
   功能说明：查询源库明细表里面已有时间戳,然后把该单据编号的最小时间戳和最大时间戳
             插入到临时表中，用完后删除该临时表的数据
   修改说明：Created BY 2011-12-12
```

```
*/
--SELECT * FROM   GRN..LSXHDMX WHERE DJBH = 'LA1000001'
IF Object_id('tempdb..#SnapShotDateTimeStamp') IS NOT NULL
BEGIN
  DROP TABLE tempdb..#SnapShotDateTimeStamp
END
CREATE TABLE #SnapShotDateTimeStamp      ----创建存放时间戳的临时表
(
    ID      INT IDENTITY(1,1) NOT NULL,
    Prevdatetime BINARY(8) NULL,
    Nextdatetime BINARY(8) NULL,
)
INSERT INTO #SnapShotDateTimeStamp
SELECT MIN(stamp),MAX(stamp) FROM   GRN..LSXHDMX WHERE DJBH = 'LA1000001'

SELECT * FROM #SnapShotDateTimeStamp

--DROP TABLE tempdb..#SnapShotDateTimeStamp   ----删除该临时表

/*
   功能说明：到追加库里面查询该时间戳之间的数据是否存在,查询匹对方案。
   修改说明: Created BY 2011-12-12
*/
SELECT * FROM   LSXHDMX WHERE DJBH = 'LA1000001'    --------查询追加库里面的时间戳

SELECT * FROM LSXHDMX,#SnapShotDateTimeStamp
WHERE stamp >= Prevdatetime AND
stamp < = Nextdatetime

/*
   功能说明：转为INT类型进行查询方案如下
   修改说明: Created BY 2011-12-12
*/
--SELECT *,CONVERT(INT,stamp) as t FROM LSXHDMX
--WHERE CONVERT(INT,stamp) > 2001
--AND CONVERT(INT ,STAMP) < 2007
```

ON 后面加 And 与 Where 区别

数据库在通过连接两张或多张表来返回记录时，都会生成一张中间的临时表，然后再将这张临时表返回给用户。

在使用left jion时，on和where条件的区别如下：

1、 on条件是在生成临时表时使用的条件，它不管on中的条件是否为真，都会返回左边表中的记录。

2、where条件是在临时表生成好后，再对临时表进行过滤的条件。这时已经没有left join的含义（必须返回左边表的记录）了，条件不为真的就全部过滤掉。

假设有两张表：

表1 tab1：

id size

1 10

2 20

3 30

表2 tab2：

size name

10 AAA

20 BBB

20 CCC

两条SQL：

1、select * form tab1 left join tab2 on (tab1.size = tab2.size) where tab2.name=’AAA’

2、select * form tab1 left join tab2 on (tab1.size = tab2.size and tab2.name=’AAA’)

第一条SQL的过程：

1、中间表

on条件：

tab1.size = tab2.size

tab1.id	tab1.size	tab2.size	tab2.name
1	10	10	AAA
2	20	20	BBB
2	20	20	CCC
3	30	(null)	(null)

2、再对中间表过滤

where 条件：

tab2.name=’AAA’

tab1.id	tab1.size	tab2.size	tab2.name
1	10	10	AAA

第二条SQL的过程：

1、中间表

on条件：

```
tab1.size = tab2.size and tab2.name='AAA'
```

(条件不为真也会返回左表中的记录)

tab1.id	tab1.size	tab2.size	tab2.name
1	10	10	AAA
2	20	(null)	(null)
3	30	(null)	(null)

其实以上结果的关键原因就是left join,right join,full join的特殊性，不管on上的条件是否为真都会返回left或right表中的记录，full则具有left和right的特性的并集。 而inner jion没这个特殊性，则条件放在on中和where中，返回的结果集是相同的。

SQLSERVER 执行顺序

SQL 不同于与其他编程语言的最明显特征是处理代码的顺序。在大数编程语言中，代码按编码顺序被处理，但是在SQL语言中，第一个被处理的子句是FROM子句，尽管SELECT语句第一个出现，但是几乎总是最后被处理。

每个步骤都会产生一个虚拟表，该虚拟表被用作下一个步骤的输入。这些虚拟表对调用者（客户端应用程序或者外部查询）不可用。只是最后一步生成的表才会返回 给调用者。如果没有在查询中指定某一子句，将跳过相应的步骤。下面是对应用于SQL server 2000和SQL Server 2005的各个逻辑步骤的简单描述。

```
(8)SELECT (9)DISTINCT  (11)<Top Num> <select list>
(1)FROM [left_table]
(3)<join_type> JOIN <right_table>
(2)       ON <join_condition>
(4)WHERE <where_condition>
(5)GROUP BY <group_by_list>
(6)WITH <CUBE | RollUP>
(7)HAVING <having_condition>
(10)ORDER BY <order_by_list>
```

逻辑查询处理阶段简介

1.FROM：对FROM子句中的前两个表执行笛卡尔积（Cartesian product)(交叉联接），生成虚拟表VT1

2.ON：对VT1应用ON筛选器。只有那些使<join_condition>为真的行才被插入VT2。

3.OUTER(JOIN)：如 果指定了OUTER JOIN（相对于CROSS JOIN 或(INNER JOIN),保留

表（preserved table：左外部联接把左表标记为保留表，右外部联接把右表标记为保留表，完全外部联接把两个表都标记为保留表）中未找到匹配的行将作为外部行添加到 VT2,生成VT3.
如果FROM子句包含两个以上的表,则对上一个联接生成的结果表和下一个表重复执行步骤1到步骤3，直到处理完所有的表为止。
4.WHERE：对VT3应用WHERE筛选器。只有使<where_condition>为true的行才被插入VT4.
5.GROUP BY：按GROUP BY子句中的列列表对VT4中的行分组，生成VT5.
6.CUBE|ROLLUP：把超组(Suppergroups)插入VT5,生成VT6.
7.HAVING：对VT6应用HAVING筛选器。只有使<having_condition>为true的组才会被插入VT7.
8.SELECT：处理SELECT列表，产生VT8.
9.DISTINCT：将重复的行从VT8中移除，产生VT9.
10.ORDER BY：将VT9中的行按ORDER BY 子句中的列列表排序，生成游标（VC10）.
11.TOP：从VC10的开始处选择指定数量或比例的行，生成表VT11,并返回调用者。
注：步骤10，按ORDER BY子句中的列列表排序上步返回的行，返回游标VC10.这一步是第一步也是唯一一步可以使用SELECT列表中的列别名的步骤。这一步不同于其它步骤的 是，它不返回有效的表，而是返回一个游标。SQL是基于集合理论的。集合不会预先对它的行排序，它只是成员的逻辑集合，成员的顺序无关紧要。对表进行排序 的查询可以返回一个对象，包含按特定物理顺序组织的行。ANSI把这种对象称为游标。理解这一步是正确理解SQL的基础。

因为这一步不返回表（而是返回游标），使用了ORDER BY子句的查询不能用作表表达式。表表达式包括：视图、内联表值函数、子查询、派生表和共用表达式。它的结果必须返回给期望得到物理记录的客户端应用程序。例如，下面的派生表查询无效，并产生一个错误：

top 和 order by 导致查询结果不一致的问题

table1结构如下：

id sort

1 1

2 2

3 2

4 2

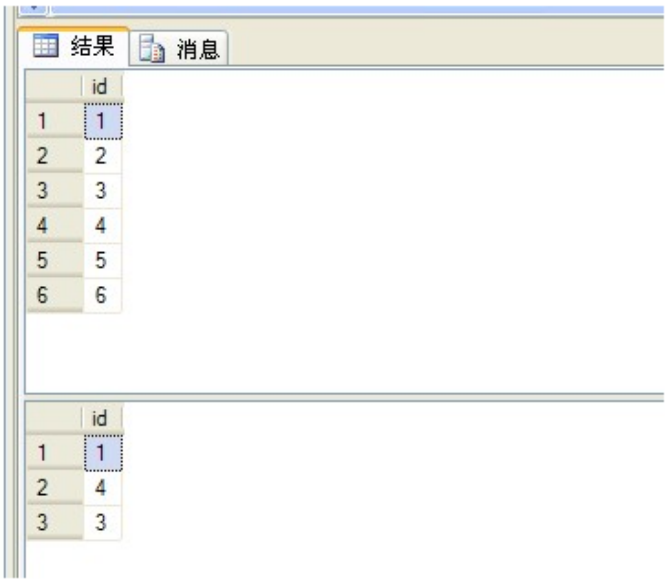
5 3

6 4

查询语句如下：

```
select id from table1 order by sort
select top 3 id from table1 order by sort  .
```

查询结果如下：



怎么top 3之后查询结果就和我们所想的“1,2,3”不一样了呢？而且我反复执行“select top 3 id from table1 order by sort”结果还是“1 4 3”，这又是为什么呢？

后来我修改了我的sql语句

```
select top 3 id from table1 order by sort,id asc
```

结果如下：

	id
1	1
2	2
3	3

正确了。

很容易可以发现，我们这里sort的值2,3,4都是相同的 原因肯定在这儿 可以看到id=4 和id=3 的时候 他是先排id=4 然后在排id=3，奇怪为什么呢？

他的排序原理是什么呢？

做一个实验

1、修改查询语句：

select top 4 id from table1 order by sort

结果如下：

	id
1	1
2	4
3	3
4	2

总结：使用top获取数据，按照sort排序时， 对待sort值相等的行-------按照id降序排序

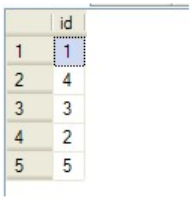
2、修改表结构：

```
id sort

1   1

2   2

3   2

4   2

5   3

6   4

7   3
```

修改查询语句：

```
select top 5 id from table1 order by sort
```

查询结果如下：



	id
1	1
2	4
3	3
4	2
5	5

可以看到id=7和id=5的时候 上述总结不成立

总结：使用top获取数据，按照sort排序时， 对待sort值相等且相邻的行-------按照id降序排序

3、验证我们刚才的总结，修改表结构：

id sort

1 1

2 2

3 2

4 2

5 3

6 4

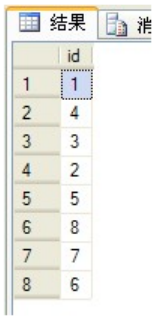
7 3

8 3

修改查询语句：

select top 8 id from table1 order by sort

查询结果如下：



结果 消

	id
1	1
2	4
3	3
4	2
5	5
6	8
7	7
8	6

结论：经验证，**使用top获取数据，按照sort排序时， 对待sort值相等且相邻的行-------按照id降序排序**

总结：同时使用top和order by时，如order by后出现值相同且相邻的行，将按照数据库表的唯一索引（主键）降序排序。

使用top和order by应注意处理order by后值相同的情况，最后手动指定能唯一标识行的字段进行做善后处理

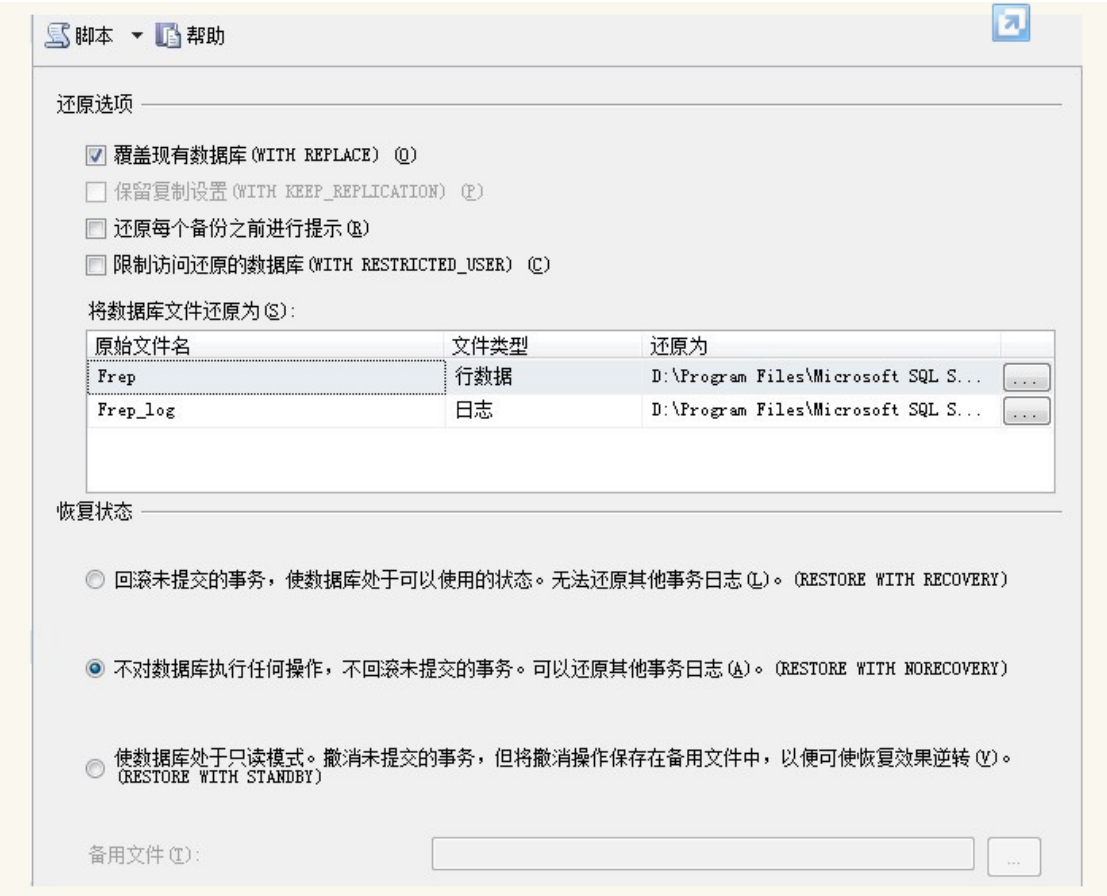
差异数据库还原问题 【SQL2008 报错：无法还原日志备份或差异备份，因为没有文件可用于前滚---还原 SQLSERVER 数据库差异备份】

其实要备份，还原最安全最有保障的是完全备份。但是完全备份肯定是需要更多的磁盘空间的开销。尤其是数据量比较大的。比如基数是500M，每天的增长量为10M，那么第一次完全备份是500M，第二次是510M,第三次是520M……成本相应的肯定就增加了很多。如果使用差异备份，那么第一次完备后的差异备份就只是第天所增长的量。能减少很多磁盘的开销。但是还原差异备份比还原完整备份稍微要麻烦一些。

还原差异备份需要一个完整备份，以及最新一个差异备份文件。
操作如下。

选择完整备份文件，在常规操作中选择“覆盖现有数据库(with replce)”与下面的单选项的第二项"[√]不对数据库执行任何操作，不回滚未提交的事务。可以还原其他事务日期（A）。（restore with norecovery"点击确定。会提示还原成功。

数据库项目文档



但是现在数据还没有还原到数据库中，数据库会一直显示为以下图显示的状态

右建再执行还原任务，选择最新的一个差异备份。上面大图所显示的操作不要选择，直接点确定就可以了。数据就完全恢复到数据库了

```
use master  RESTORE DATABASE [AAA] FROM DISK = N'D:\ceshi\AAA.BAK' WITH FILE = 1, MOVE N'AAAA' TO N'D:\DB\AAA.mdf', MOVE N'AAAA_log' TO N'D:\DB\AAA_1.ldf', NORECOVERY, NOUNLOAD, REPLACE, STATS = 10  RESTORE DATABASE [AAA] FROM  DISK = N'D:\ceshi\AAAA01.BAK' WITH  FILE = 1, NOUNLOAD,  STATS = 10
```

SQLSERVER 计算列的使用

计算列区别于需要我们手动或者程序给予赋值的列,它的值来源于该表中其它列的计算值。比如，一个表中包含有数量列Number与单价列Price,我们就可以创建计算列金额Amount来表示数量*单价的结果值，创建Amount列后，在程序中需要使用计算金额这个值时，就不用取出

Number列与Price列的值后相乘，而是直接取Amount列的值就可以了。

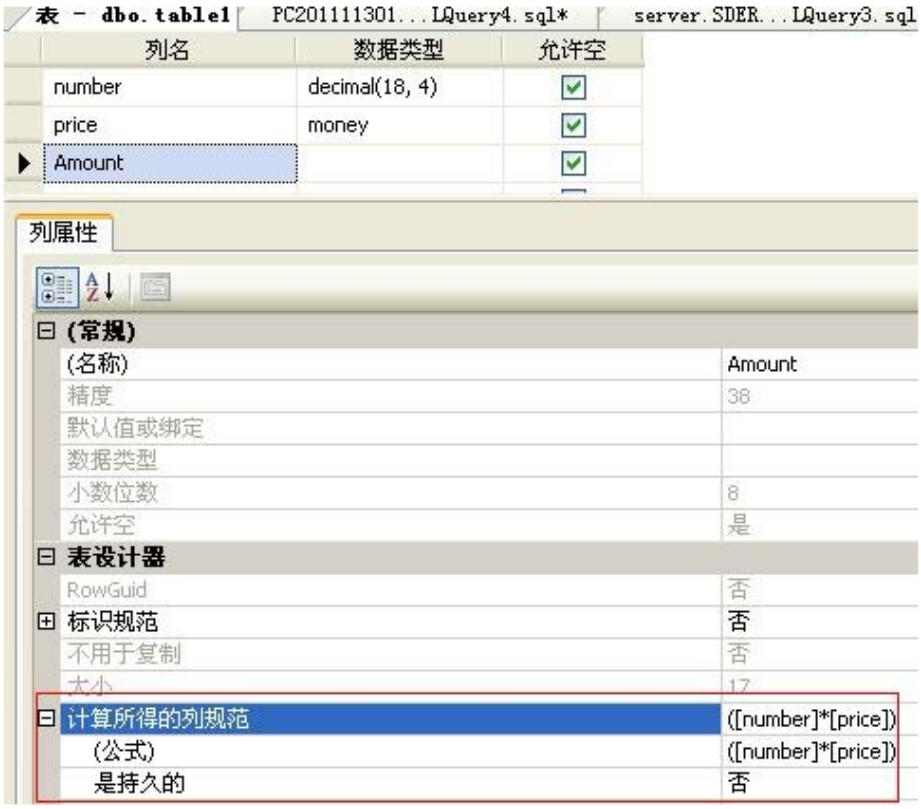
那么这个计算列要如何建立呢?

先看通过sql的方法创建:

```
create table table1
(
number decimal(18,4),
price money,
Amount as number*price --这里就是计算列
)
```

计算列是不需要我们指定数据类型与是否允许为null等信息的，SqlServer会根据情况自动赋予数据类型。

在microsoft sql server management studio建创计算列更是简单。如下图所示，只要在列属性中的"计算所得的列规范"-"公式"中填写计算列的公式就可以了



计算列的修改方式如下，必须先删除

```
alter table BB drop column CC
alter table BB add CC as 3
```

动态 SQL 执行的结果赋值给变量

```
 /*
 功能说明：动态执行SQL前面一定要加一个N字才可以的，注意语句的正确使用及开发
*/

DECLARE @TableName           VARCHAR(50),
       @FiledName           VARCHAR(50),
       @PFSQL               NVARCHAR(500),
       @PFTSQL              VARCHAR(500),
       @AutoPartitionPosition INT = 3;

SELECT @TableName = 'SG_Gathering',
      @FiledName = 'Vshop';

SET @PFSQL = '
SELECT @PFTSQL = Stuff((SELECT ''''''','''''' + Name
                    FROM   (

                    SELECT DISTINCT LEFT(' + @FiledName + ', ' +
Ltrim(@AutoPartitionPosition) + ') AS NAME
                           FROM   ' + @TableName + '

                           ) A
                    FOR XML PATH('''')), 1, 2, '''')+''''''''
                    '

EXEC   SP_EXECUTESQL   @PFSQL,N'@PFTSQL   VARCHAR(500)   output
',@PFTSQL OUTPUT

SELECT @PFTSQL
```

解决事务日志已满的问题

响应已满事务日志的备选方法包括：

备份日志。

释放磁盘空间以便日志可以自动增长。

将日志文件移到具有足够空间的磁盘驱动器。

增加日志文件的大小。

在其他磁盘上添加日志文件。

完成或取消长时间运行的事务。

备份日志： 在完整恢复模式或大容量日志恢复模式下，如果最近尚未备份事务日志，则请立即进行备份以免发生日志截断。 如果从未备份日志，则必须创建两个日志备份，以允许数据库引擎将日志截断到上次的备份点。 截断日志可释放空间以供新的日志记录使用。 若要防止日志再次填满，请经常执行日志备份。

备份事务日志功能：

```
BACKUP LOG [BaisonSDM] TO [BaisonSDM] WITH NOFORMAT, NOINIT, NAME =
N'BaisonSDM-事务日志 备份', SKIP, NOREWIND, NOUNLOAD, STATS = 10
```

Merge 的概念的介绍及说明

http://www.cnblogs.com/lenxu/archive/2012/02/14/2350922.html (Merge同步)
http://www.cnblogs.com/xfrog/archive/2010/10/13/1850602.html [Merge功能使用]
用途
merge 命令可以用来用一个表中的数据来修改或者插入到另一个表。插入或者修改的操作取决于on子句的条件。

该语句可以在同一语句中执行两步操作，可以减少执行多条insert 和update语句。

merge是一个确定性的语句，即不会在同一条merge语句中去对同一条记录多次做修改操作。

语法

其中，merge_insert_clause::=

关键字、参数

into 子句

在into子句中指定所要修改或者插入数据的目标表

using 子句
在using子句中指定用来修改或者插入的数据源。数据源可以是表、视图或者一个子查询语句。

on 子句
在on子句中指定执行插入或者修改的满足条件。在目标表中符合条件的每一行，oracle用数据源中的相应数据修改这些行。对于不满足条件的那些行，oracle则插入数据源中相应数据。

when matched | not matched
用该子句通知oracle如何对满足或不满足条件的结果做出相应的操作。可以使用以下的两类子句。

merge_update子句
merge_update子句执行对目标表中的字段值修改。当在符合on子句条件的情况下执行。如果修改子句执行，则目标表上的修改触发器将被触发。

限制：当修改一个视图时，不能指定一个default值

merge_insert 子句
merge_insert子句执行当不符合on子句条件时，往目标表中插入数据。如果插入子句执行，则目标表上插入触发器将被触发。

限制：当修改一个视图时，不能指定一个default值

范例
merge 范例

```
merge into bonuses d

using (select employee_id, salary, department_id from employees
```

```
where department_id = 80) s

on (d.employee_id = s.employee_id)

when matched then update set d.bonus = d.bonus + s.salary*.01

when not matched then insert (d.employee_id, d.bonus)

values (s.employee_id, s.salary*0.01);
```

Merge 的使用(数据同步，数据转换，基于目标表 INSERT)

http://www.cnblogs.com/careyson/archive/2012/03/07/2383690.html(Merge 同步)

```
/*
   功能说明:Merge简介

   备注： Merge关键字是一个神奇的DML关键字。它在SQL Server 2008被引入，它能将Insert,Update,Delete简单的并为一句。MSDN对于Merge的解释非常的短小精悍:”根据与源表联接的结果，对目标表执行插入、更新或删除操作。例如，根据在另一个表中找到的差异在一个表中插入、更新或删除行，可以对两个表进行同步。”,通过这个描述，我们可以看出Merge是关于对于两个表之间的数据进行操作的。

可以想象出，需要使用Merge的场景比如:
•    数据同步
•    数据转换
•    基于源表对目标表做Insert,Update,Delete操作

使用Merge关键字的好处：首先是更加短小精悍的语句，在SQL Server 2008之前没有Merge的时代，基于源表对目标表进行操作需要分别写好几条Insert,Update,Delete。而使用Merge,仅仅需要使用一条语句就好。下面我们来看一个例子。

    首先建立源表和目标表，并插入相关的数据,如图所示。

*/
--创建源表
```

数据库项目文档

```
IF EXISTS (SELECT 1
          FROM   sysobjects
          WHERE  id = Object_id('SourceTable')
                 AND type = 'U')
  BEGIN
      DROP TABLE SourceTable
  END

GO

CREATE TABLE SourceTable
  (
     id         INT,
     descdetail VARCHAR(50)
  )

--创建目标表
IF EXISTS (SELECT 1
          FROM   sysobjects
          WHERE  id = Object_id('TargetTable')
                 AND type = 'U')
  BEGIN
      DROP TABLE TargetTable
  END

GO

CREATE TABLE TargetTable
  (
     id         INT,
     descdetail VARCHAR(50)
  )

---为源表插入数据
INSERT INTO SourceTable
           (id,
            descdetail)
SELECT 1,
       '描述'
UNION ALL
SELECT 2,
```

```
      '描述'
UNION ALL
SELECT 3,
      '描述'
UNION ALL
SELECT 4,
      '描述'

---为目标表插入数据
INSERT INTO TargetTable
           (id,
            descdetail)
SELECT 1,
      '在源表里存在，将会被更新'
UNION ALL
SELECT 2,
      '在源表里存在，将会被更新'
UNION ALL
SELECT 5,
      '在源表里不存在，将会被删除'
UNION ALL
SELECT 6,
      '在源表里不存在，将会被删除'

/*
   功能说明:下面我们来写一个简单的Merge语句，如图所示
*/
MERGE INTO TargetTable AS T
USING SourceTable AS S
ON T.ID = S.ID
WHEN MATCHED --当上面On后面的T.id=s.id时，目标表中的ID为,2的数据被更新
THEN
  UPDATE SET T.DESCDETAIL = S.DESCDETAIL
WHEN NOT MATCHED ---目标表中没有的ID,在源表中有，则插入相关数据
THEN
  INSERT
  VALUES(s.id,
         s.descdetail)
WHEN NOT matched BY source --目标表中存在，源表中不存在，则删除
THEN
  DELETE;
```

```
SELECT *
FROM   SourceTable

SELECT *
FROM   TargetTable

/*
 功能说明:Merge语句还有一个强大的功能是通过OUTPUT子句，可以将刚刚做过变动的数据进行输出。我们在上面的Merge语句后加入OUTPUT子句,如图所示。
*/
MERGE INTO TargetTable AS T
USING SourceTable AS S
ON T.ID = S.ID
WHEN MATCHED --当上面On后面的T.id=s.id时，目标表中的ID为,2的数据被更新
THEN
  UPDATE SET T.DESCDETAIL = S.DESCDETAIL
WHEN NOT MATCHED ---目标表中没有的ID,在源表中有，则插入相关数据
THEN
  INSERT
  VALUES(s.id,
         s.descdetail)
WHEN NOT matched BY source --目标表中存在，源表中不存在，则删除
THEN
  DELETE

OUTPUT $ACTION AS [ACTION],Inserted.id AS 插入的id,Inserted.descdetail AS 插入的descdetail,
deleted.id AS 删除的id,deleted.descdetail AS 删除的descdetail;

/*
功能说明:当然了，上面的Merge关键字后面使用了多个WHEN…THEN语句，而这个语句是可选的.也可以仅仅新增或是仅仅删除,如图所示
*/
MERGE INTO TargetTable AS T
USING SourceTable AS S
ON T.ID = S.ID
WHEN NOT MATCHED ---目标表中没有的ID,在源表中有，则插入相关数据
THEN
  INSERT
  VALUES(s.id,
```

```
        s.descdetail)

OUTPUT $ACTION AS [ACTION],Inserted.id AS 插入的id,Inserted.descdetail AS
插入的descdetail,
deleted.id AS 删除的id,deleted.descdetail AS 删除的descdetail;

/*
 功能说明：我们还可以使用TOP关键字限制目标表被操作的行，如图所示。在图的语句基础上
加上了TOP关键字，我们看到只有两行被更新
*/
MERGE TOP(2) TargetTable AS T
USING SourceTable AS S
ON T.ID = S.ID
WHEN MATCHED --当上面On后面的T.id=s.id时，目标表中的ID为,2的数据被更新
THEN
  UPDATE SET T.DESCDETAIL = S.DESCDETAIL
WHEN NOT MATCHED ---目标表中没有的ID,在源表中有，则插入相关数据
THEN
  INSERT
  VALUES(s.id,
        s.descdetail)
WHEN NOT matched BY source --目标表中存在，源表中不存在，则删除
THEN
  DELETE

OUTPUT $ACTION AS [ACTION],Inserted.id AS 插入的id,Inserted.descdetail AS
插入的descdetail,
deleted.id AS 删除的id,deleted.descdetail AS 删除的descdetail;

/*
功能说明:仅仅是MATCHED这种限制条件往往不能满足实际需求，我们可以在图那个语句的基础
上加上AND附加上额外的限制条件，如图所示。
*/
MERGE INTO TargetTable AS T
USING SourceTable AS S
ON T.ID = S.ID AND S.ID =3  --加入ID=3的限定条件
WHEN NOT MATCHED ---目标表中没有的ID,在源表中有，则插入相关数据
THEN
  INSERT
  VALUES(s.id,
        s.descdetail)
```

```
OUTPUT $ACTION AS [ACTION],Inserted.id AS 插入的id,Inserted.descdetail AS 插入的descdetail,
deleted.id AS 删除的id,deleted.descdetail AS 删除的descdetail;

/*
Merge关键字的一些限制
•  使用Merge关键字只能更新一个表
•  源表中不能有重复的记录


小结
    本文简单说明了Merge关键的字的使用。如果你使用的是SQL Server 2008之后的版本，在面对一些比如库存结账之类的业务时，放弃IF…ELSE和手写UPDATE，Insert吧，使用Merge关键字可以使这类操作更加轻松愉悦。

*/
```

把 1,2,3,4 转换成 IN 所需要的

```
 /*
 功能说明：字符串转换
*/
 DECLARE @SQL VARCHAR(50)='1,2,3,4';
 SELECT ''''+REPLACE(@SQL,',',''',''')+''''
```

获取明细表中 重复的数据

```
/*
  功能说明：获取明细表中重复的数据
  修改说明：Created BY LY 2013-1-5
*/
SELECT OM_ORDERS.*
FROM   OM_ORDERS
WHERE  OM_ORDERS.OD_STATE <> 20
      AND ( OD_ID IN (SELECT ODD_ORDER_ID
                      FROM   OM_ORDER_DETAILS a
                      WHERE  ODD_ORDER_ID NOT IN (SELECT ODD_ORDER_ID
                                                  FROM   OM_ORDER_DETAILS a
                                                  WHERE  ODD_STATE <> 20)) )
```

```
except

SELECT *
FROM   OM_ORDERS
WHERE  OD_ID NOT IN(SELECT ODD_ORDER_ID
                    FROM   OM_ORDER_DETAILS
                    WHERE  ODD_STATE <> 20)
       AND OD_STATE <> 20



select   *
from OM_ORDER_DETAILS a
where ODD_STATE not in (
select
    ODD_STATE
from OM_ORDER_DETAILS a
where ODD_STATE <> 20)
```

执行动态 SQL 语句，并返回参数

```
/*
  功能说明：执行动态SQL语句，并返回参数
  修改说明:Created BY LY 2013-1-8
*/
DECLARE @sql     NVARCHAR(500),
        @str     NVARCHAR(500),
        @outstr  NVARCHAR(500),
        @sqltest NVARCHAR(500);

SELECT @str = 'select  @outstr = replace(WI_RECEIVE_LIST,''"'','''') from
WF_WORKFLOW_INST where WI_ID=1444';

EXEC Sp_executesql
  @str,
  N'@outstr nvarchar(500) output',
  @sqltest output;

SELECT @sqltest
```

```
SET @sql ='
 select USER_LOGIN_NAME from UR_USERS where USER_ID in
 (' + @sqltest + ')'

EXEC (@sql)
```

常用递归的应用(CTE)

```
DECLARE @TABLE TABLE (
  ID       INT NULL,
  ParentID INT NULL )

INSERT INTO @TABLE
SELECT 10000,
       -1
UNION ALL
SELECT 13551,
       10000
UNION ALL
SELECT 15,
       13551
UNION ALL
SELECT 245,
       15
UNION ALL
SELECT 5555,
       10000
UNION ALL
SELECT 88,
       5555
UNION ALL
SELECT 255,
       88
UNION ALL
SELECT 6666,
       10000
UNION ALL
SELECT 999,
```

```
      6666
UNION ALL
SELECT 444,
      999
UNION ALL
SELECT 278,
      444;

SELECT * FROM @TABLE;
WITH MU
    AS (SELECT ID,
               PARENTID,
               CONVERT(VARCHAR(MAX), ID) AS PATH,
               1                         AS LEVEL
        FROM   @TABLE
        WHERE  PARENTID = -1    --固定变量
        UNION ALL
        SELECT T1.ID,
               T1.PARENTID,
               T2.PATH + ',' + Ltrim(T1.ID),
               LEVEL + 1
        FROM   @TABLE T1
               INNER JOIN MU T2
                 ON T1.PARENTID = T2.ID)
SELECT CASE
        WHEN LEVEL > 2 THEN LEFT(Stuff(PATH, 1, Charindex(',', PATH), ''),
Charindex(',', Stuff(PATH, 1, Charindex(',', PATH), '')) - 1)
        WHEN LEVEL = 2 THEN ID
        ELSE NULL
      END AS '第二层节点',
      *
FROM   MU
```

多个 Update 联合使用进行修改完善

```
UPDATE ZWSKC
SET    ZWSKC.QDDBSL = B.SL,
      ZWSKC.QDDBJE = B.SL * dbo.ZWSKC.QMJE
FROM   ZWSKC,
```

```
       (SELECT DB.SPDM,
               Sum(DB.SL) AS SL
        FROM   VW_QDDBD DB,
               dbo.QUDAO QD
               LEFT JOIN dbo.QUDAO_A
                 ON QD.QDDM = dbo.QUDAO_A.QDDM
        WHERE  DB.QDDM = QD.QDDM
               AND Isnull(dbo.QUDAO_A.DLHS, '0') = '1'
               AND QD.SJQD = '000'
               AND DB.RQ >= '2010-12-1'
               AND DB.RQ <= '2010-12-13'
        GROUP  BY DB.SPDM) B
WHERE  ZWSKC.SPDM = B.SPDM
       AND dbo.ZWSKC.HJMN = '201012'
```

临时表不能做动态 SQL

```
DECLARE @SPMX_QM TABLE (SPDM varchar(20),SL numeric(18,2), JE
numeric(18,4));
DECLARE @SQL NVARCHAR(500);
SET @SQL = 'INSERT INTO @SPMX_QM
SELECT ''AFD'',11,22';
exec (@SQL)

SELECT * FROM @SPMX_QM
```

动态 SQL 插入到临时表

```
create table T_Sales_Order
(
id int,
createdate datetime
)
go
create table T_Sales_Product
(
parentid int,
BaseQuantity int
)
```

```
go
if object_id('tempdb.dbo.#TotalSalesForMonth') is not null
    drop table #TotalSalesForMonth  --删除临时表
select * into  #TotalSalesForMonth
from (
  select
 cast(month(CreateDate) as nvarchar)   as   monthN,
    sum(BaseQuantity)   as   total
  from   (
 select a.CreateDate,b.BaseQuantity
 from  dbo.T_Sales_Order a left join dbo.T_Sales_Product as b
 on a.Id =b.ParentId
 where year(a.CreateDate)=year(getdate())
  ) c
  group by month(CreateDate)
) p
DECLARE @str VARCHAR(500) ,@Sql NVARCHAR(max)
SET @str=''
SELECT @str=@str+','+'['+cast(monthN as nvarchar)+']' FROM #TotalSalesForMonth
SET @str=right(@str,len(@str)-1)
--由于 pivot 不支持 在 in ()中直接添加字符串，所以需要使用 sql 拼接一下

SET @Sql='select * into #ForMonth  from  #TotalSalesForMonth pivot (sum(total)
for monthN in ('+@str+') ) as pvt '
if object_id('tempdb.dbo.#ForMonth') is not null
    drop table #ForMonth  --删除临时表
exec(@Sql)
```

2.SQLSERVER2005/2008 的新语法操作。

CTE 的使用

公用表表达式（CTE）具有一个重要的优点，那就是能够引用其自身，从而创建递归 CTE。递归 CTE 是一个重复执行初始 CTE 以返回数据子集直到获取完整结果集的公用表表达式。
在 SQL Server 2005 中，当某个查询引用递归 CTE 时，它即被称为“递归查询”。递归查询通常用于返回分层数据，例如：显示某个组织图中的雇员或物料清单方案（其中父级产品有一

个或多个组件，而那些组件可能还有子组件，或者是其他父级产品的组件）中的数据。
递归 CTE 可以极大地简化在 SELECT、INSERT、UPDATE、DELETE 或 CREATE VIEW 语句中运行递归查询所需的代码。在 SQL Server 的早期版本中，递归查询通常需要使用临时表、游标和逻辑来控制递归步骤流。有关公用表表达式的详细信息，请参阅使用公用表表达式。
CONVERT是专对SQL Server使用的，使日期与时间值，小数之间转换具有更宽的灵活性。
CAST是两种功能中更具ANSI标准的功能，即虽然更具便携性（比如，使用CAST的函数能更容易的被其它数据库软件使用），但功能相对弱一些。

一、用 CTE 删除重复数据

```
/*
用CTE处理重复数据-删除重复数据
作者：梁勇
时间：-12-7

ID          Num         Name
----------- ----------- ----
1           1           A
2           1           A
3           2           B
4           2           B

转为：

ID          Num         Name
----------- ----------- ----
1           1           A
3           2           B
*/
use City;
go
set nocount on
if not object_id('A') is null
    drop table A
Go
Create table A
(
ID int identity(1,1) primary key not null,
Num int,
Name nvarchar(1)
)
Insert into A
```

```
select 1,N'A' union all
select 1,N'A' union all
select 2,N'B' union all
select 2,N'B'
go
select * from A;
with cte as
(
select ROW_NUMBER()over(partition by Num,Name order by Getdate()) as RN
from A
)
delete from cte where RN>1
select * from A
drop table A
```

二、用 CTE 实现递归

```
/*
名称       分类
tt
aa
bb          aa
cc          bb
dd          cc
ee          dd
mm          tt
无限分类...

要输出名称='ee'上层所有的名称

正确输出答案如下：（要求从最上层到下层）

aa-bb-cc-dd
*/
---创建表
if OBJECT_ID(N'Tbl1',N'U') is not null drop table Tbl1
go
create table Tbl1
(
  名称varchar(10) not null,
  分类varchar(10)
)
go
```

数据库项目文档

```
insert into  Tbl1
select 'tt','' union all
select 'aa','' union all
select 'bb','aa' union all
select 'cc','bb' union all
select 'dd','cc' union all
select 'ee','dd' union all
select 'mm','tt'
declare @a nvarchar(500);

-------cte递归
with cte as
(
    SELECT 名称,分类FROM Tbl1 where 名称= 'ee'   --固定变量
    UNION ALL
    select a.名称,a.分类FROM Tbl1 a
    inner join cte b on a.名称= b.分类
)
select stuff(
           (select '-'+名称from cte where 名称<> 'ee' order by 名称for xml
path('')
           ),1,1,''
          ) as '输出'

--select @a = case isnull(@a, '') when '' then '' else @a + '-' end + 名
称from
--cte order by 名称
--select @a
--select '-'+名称from cte order by 名称for xml path('')
--select * from cte where 名称<> 'ee'
三、利用递归查询得到一个部门的树
/*
重点：1、[递归成员]中[关键标示符*10+新产生的关键标示符
         (row_number()over(orderby getdate()))]
      2、最后order by ltrim(标示符),因为整型+字符型=整型，根据字符串排序
作者：梁勇
时间：2010-12-7
描述：输入一定的数据，输出一个树形数据。
延伸：可以动态获取某个部门下的所以子部门。也可以获取该部门上级的所以部门
递归查询中：第二条记录可以引用第一条记录的值
```

数据库项目文档

```
ID          Name                 Parent
----------- -------------------- --------------------
1           大学                 NULL
2           学院                 大学
3           计算机学院           学院
4           网络工程             计算机学院
5           信息管理             计算机学院
6           电信学院             学院
7           教务处               大学
8           材料科               教务处
9           招生办               大学
10          教育方向             信息管理
```

转换为：

```
TE
--------------------------------
大学
  |_学院
      |_计算机学院
            |_网络工程
            |_信息管理
                  |_教育方向
      |_电信学院
  |_教务处
       |_材料科
  |_招生办

*/
use City;
go
if OBJECT_ID(N'Tree',N'U') is not null drop table Tree
create table Tree
(
  ID int identity(1,1) primary key not null,
  Name varchar(20) not null,
  Parent varchar(20) null
)
go
insert Tree values('大学',null)
```

```
insert Tree values('学院','大学')
insert Tree values('计算机学院','学院')
insert Tree values('网络工程','计算机学院')
insert Tree values('信息管理','计算机学院')
insert Tree values('电信学院','学院')
insert Tree values('教务处','大学')
insert Tree values('材料科','教务处')
insert Tree values('招生办','大学')
insert Tree values('教育方向','信息管理')
go
select * from Tree;
with cte as
(
select cast(Name as nvarchar(100)) as TE,Name,Parent,0 as
Levle,ROW_NUMBER()over(order by getdate()) as RN from Tree where Parent
is null
union all
select cast(replicate(' ',len(cte.TE))+'|_'+Tree.Name as nvarchar(100))
as TE,Tree.Name,Tree.Parent,cte.Levle+1 as
Levle,cte.RN*10+ROW_NUMBER()over(order by getdate())as RN from Tree
inner join cte
on Tree.Parent=cte.Name
)
select * from cte order by ltrim(RN)
drop Table Tree

--1.将 CTE 表达式拆分为定位点成员和递归成员。
--2.运行定位点成员，创建第一个调用或基准结果集 (T0)。
--3.运行递归成员，将 Ti 作为输入(这里只有一条记录)，将 Ti+1 作为输出。
--4.重复步骤 3，直到返回空集。
--5.返回结果集。这是对 T0 到 Tn 执行 UNION ALL 的结果。
```

窗口函数

窗口函数和聚集函数一样都是对定义的行集（组）进行聚集，但是不像聚集一样只返回一个值，窗口函数可以为每个组返回多个值，执行聚集的行组是窗口(因此称为‘窗口函数’)。窗口函数是在聚集函数的基础上加了一个 over()，所有的聚集函数都可以利用这种方式转换成窗口函数。窗口函数是最后才执行的，在order by 之前,where和group by之后

	ID	ProvinceName	CityName	CityPopulation
1	26	福建	南平	20
2	27	福建	三明	50
3	28	福建	泉州	110
4	31	湖南	长沙	10
5	32	湖南	常德	30
6	34	浙江	嘉兴	20

图1

	记录数
1	6

图2

	ProvinceName	CityName	记录数
1	福建	南平	6
2	福建	三明	6
3	福建	泉州	6
4	湖南	长沙	6
5	湖南	常德	6
6	浙江	嘉兴	6

图3

图1表示是原数据表的数据，SQL语句：`select * from CityTable`

图2是获取原数据表中的记录数，SQL语句：`select count(*)as '记录数' from CityTable`

图3是根据窗口函数 `count(*) over()`获取数据表记录数，SQL语句：`select ProvinceName,CityName, count(*) over() as '记录数' from CityTable`。

由上面的比较可以得出，利用聚集函数`count(*)`只能得到一行记录(如图2所示)，这条记录包含聚集后的结果6，而用窗口函数`count(*) over()`得到的记录不只一行(如图3所示)，而是多行，且每行都包含了聚集结果。而这也是聚集函数和窗口函数最大的区别。

当然上面的over关键字后面的()中还可以接受其他子句，以改变窗口函数所作用的行范围，如果over()括号中没有任何子句，那么默认窗口函数作用于整个结果集。

分区函数

`[Partition By]`分区子句：可以根据`partition by`子句定义行的分区或组，以完成聚集，如果使用空括号，那么整个结果集就是分区，窗口函数将对它进行聚集计算，可以把`Partition By`看成是移动的`Group By`，可以用`Partition By`对定义的行组计算聚集(当遇到新的组时复位)，并返回每个值(每个组中的成员)，而不是用一个组表示表中这个值的所有实例。例如将图3所示的结果集根据省份不同进行分区，SQL语句如下：

```
select ProvinceName,CityName,
count(*) over(partition by ProvinceName) as '记录数'
from CityTable
```

上面涉及到一个函数`prtition by ProvinceName`，意思是将整个结果集进行分区，根据不同的省份划分不同的分区，并在结果集中显示每个分区中记录行的个数，当遇到新的分区时再进行重新计算。执行后的如图4所示

	ProvinceName	CityName	记录数
1	福建	南平	3
2	福建	三明	3
3	福建	泉州	3
4	湖南	长沙	2
5	湖南	常德	2
6	浙江	嘉兴	1

图4

窗口函数是最后才执行的，在order by 之前,where和group by之后。举个例子，还是用到图4结果集的SQL语句，外加一个条件

```
where (ProvinceName='福建' and CityName<>'南平') or (ProvinceName='湖南' and CityName<>'长沙') order by '记录数'
```

最后的SQL语句如下

```
select ProvinceName,CityName,
count(*) over(partition by ProvinceName) as '记录数'
from CityTable
where (ProvinceName='福建' and CityName<>'南平') or (ProvinceName='湖南' and CityName<>'长沙')
order by '记录数'
```

上面SQL首先根据where语句获取符合条件的记录，然后再执行窗口函数进行计数(此时湖南对应的记录数是1，而不是图4所示的2，福建对应的记录数为2,也不是图4所示的3)，最后再根据order by语句进行排序，最后显示结果如图5所示

	ProvinceName	CityName	记录数
1	湖南	常德	1
2	福建	三明	2
3	福建	泉州	2

图5

另附count(*)和count(column)的区别

count(*)计算的是行的数目，而count(column)计算的是列column值非空的记录数，即count(column)是忽略空值的，那么当空值也作为有效值时，我们就只能用count(*)计算行数来进行聚集了。

Pivot

可以使用 PIVOT 和 UNPIVOT 关系运算符将表值表达式更改为另一个表。PIVOT 通过将表达式某一列中的唯一值转换为输出中的多个列来旋转表值表达式,并在必要时对最终输出中所需的任何其余列值执行聚合。UNPIVOT 与 PIVOT 执行相反的操作，将表值表达式的列转换为列值。

PIVOT 提供的语法比一系列复杂的 SELECT...CASE 语句中所指定的语法更简单和更具

可读性。有关 PIVOT 语法的完整说明，请参阅 FROM (Transact-SQL)。

以下是带批注的 PIVOT 语法。

```
SELECT <非透视的列>,
    [第一个透视的列] AS <列名称>,
    [第二个透视的列] AS <列名称>,
    ...
    [最后一个透视的列] AS <列名称>,
FROM
    (<生成数据的 SELECT 查询>)
    AS <源查询的别名>
PIVOT
(
    <聚合函数>(<要聚合的列>)
FOR
[<包含要成为列标题的值的列>]
    IN ( [第一个透视的列], [第二个透视的列],
    ... [最后一个透视的列])
) AS <透视表的别名>
<可选的 ORDER BY 子句>;
USE AdventureWorks2008R2;
GO
SELECT VendorID, [250] AS Emp1, [251] AS Emp2, [256] AS Emp3, [257] AS Emp4, [260] AS Emp5
```

```
    FROM
    (SELECT PurchaseOrderID, EmployeeID, VendorID
    FROM Purchasing.PurchaseOrderHeader) p
    PIVOT
    (
    COUNT (PurchaseOrderID)
    FOR EmployeeID IN
    ( [250], [251], [256], [257], [260] )
    ) AS pvt
    ORDER BY pvt.VendorID;
```

```
create table tb(姓名 varchar(10) , 课程 varchar(10) , 分数 int)
insert into tb values('张三' , '语文' , 74)
insert into tb values('张三' , '数学' , 83)
insert into tb values('张三' , '物理' , 93)
insert into tb values('李四' , '语文' , 74)
insert into tb values('李四' , '数学' , 84)
insert into tb values('李四' , '物理' , 94)
go
```

```
--SQL SERVER 2005 静态SQL。
select * from (select * from tb) a pivot (max(分数) for 课程 in (语文,数学,物理)) b
```

UnPivot

UNPIVOT 将与 PIVOT 执行几乎完全相反的操作，将列转换为行。假设以上示例中生成的表在数据库中存储为 pvt，并且您需要将列标识符 Emp1、Emp2、Emp3、Emp4 和 Emp5 旋转为对应于特定供应商的行值。这意味着必须标识另外两个列。包含要旋转的列值（Emp1、Emp2...）的列将被称为 Employee，将保存当前位于待旋转列下的值的列被称为 Orders。这些列分别对应于 Transact-SQL 定义中的 pivot_column 和 value_column。以下为该查询。

例如：

```
--Create the table and insert values as portrayed in the previous example.
    CREATE TABLE pvt (VendorID int, Emp1 int, Emp2 int,
        Emp3 int, Emp4 int, Emp5 int);
    GO
```

```
    INSERT INTO pvt VALUES (1,4,3,5,4,4);
    INSERT INTO pvt VALUES (2,4,1,5,5,5);
    INSERT INTO pvt VALUES (3,4,3,5,4,4);
    INSERT INTO pvt VALUES (4,4,2,5,5,4);
    INSERT INTO pvt VALUES (5,5,1,5,5,5);
    GO
    --Unpivot the table.
    SELECT VendorID, Employee, Orders
    FROM
       (SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp5
       FROM pvt) p
    UNPIVOT
       (Orders FOR Employee IN
          (Emp1, Emp2, Emp3, Emp4, Emp5)
    )AS unpvt;
GO
```

For xml Path

```
DECLARE @x XML
SELECT @x = '
<People>
    <dongsheng type="One" Title="News">
        <Info Name="Email">One Email@11.com</Info>
        <Info Name="Phone">One 15886643898</Info>
        <Info Name="Email">One Email@22.com</Info>
    </dongsheng>
    <dongsheng type="Two">
        <Info Name="Email">Two Email@11.com</Info>
        <Info Name="Phone">Two Phone</Info>
        <Info Name="Email">Two Email@55.com</Info>
    </dongsheng>
    <dongsheng type="Two" Dep="IT">
        <Info Name="Email">Two-Email@11.com-IT</Info>
        <Info Name="Phone">Two-Phone-IT</Info>
        <Info Name="Email2">Two-Email@55.com-IT</Info>
    </dongsheng>
    <dongsheng type="Two" Dep="IT" Title="News">
    </dongsheng>
```

```
</People>'

/*
描 述：定位并获取节点的值和节点属性值
注意点：
1、如果要获取两列或以上用第一种更简洁，若只获取一列上诉两种方法都行。但是第二种跟直观
2、当这个路径下面获取到的nodes或者value不止时，若要获取第i个元素，那么
   要用()符号将整个结果集包含进去，然后后跟[i]即可
*/
--方法一
select                                                          --1
   C.value('.','nvarchar(30)') as MyEmail,                          --2获取节点值
   C.value('(@Name)[1]','nvarchar(20)') as MyPhone--获取属性
from @x.nodes('(/People/dongsheng[@type="Two"][@Dep="IT"])[1]/Info') T(C)--3

--方法二
select

@x.value('(//People/dongsheng[@type="Two"][@Dep="IT"])[1]/Info[@Name="Email"])[1]','nvarchar(20)') as MyEmail,--获取节点值

@x.value('((//People/dongsheng[@type="Two"][@Dep="IT"])[1]/Info)[1]/@Name)','nvarchar(20)') as MyPhone--获取属性
 --
 /*
 运行顺序：
 1、执行语句(找节点)。即执行上面才from.根据方法.nodes找到节点。以/开头，T(C)是一个表值方法，C估计是其列，
    >>找到dongsheng这个节点
 2、执行语句(访问值)。<instance of xml data type>.value(<xpath location>,<non-xml sql server type>)
    >>C引用的就是dongsheng这个节点，然后继续访问下去。访问那些其属性Name="Email"的info节点，
    注意点：、访问属性要以@开头，访问到的节点要以()括号起来，然后跟上索引[i]来访问其中的哪个节点
                (可能结果集不止一个)
            2、要访问一个结果集中的某个就用上面的sql语句。如果要访问所有符合条件的节点则用下面的
```

```
            select
              C.value('.','nvarchar(30)') as my8 --这里只能是'.'
            from @x.nodes('/People/dongsheng/Info[@Name="Email"]')
T(C) --这里访问到最后的节点
  3、执行语句
  4、文档上说.value必须返回标量值，即只能一行，而且这个值要转换成一种非xml的类型
  5、在语句中的'.'一般情况是出现在语句中已经找到目标元素，然后在语句中想用C.value
获取其完整值
 */


 /*
 set @XmlInstance.modify()方法的使用
 1、insert          添加(节点、属性)
 2、delete          删除(节点、属性)
 3、replace value of 修改(节点值、属性值)
 */
 --1、Update Attribute-Value
 declare @Name nvarchar(20)
 declare @X1 xml
 select  @X1='<Info Name="Phone">One 15886643898</Info>'
 select @X1
 select @Name='huangzhubin'
 set @X1.modify(' replace value of (/Info/@Name)[1]
              with sql:variable("@Name") ')
 select @X1

--1、Update Element-Value
DECLARE @xUpdate XML
SELECT @xUpdate = '
<Peoples>
  <People>
      <NAME>土豆</NAME>
      <SEX>男</SEX>
      <QQ>5345454554</QQ>
  </People>
</Peoples>'
SELECT @xUpdate
DECLARE @SEXUpdate CHAR(2)
SELECT @SEXUpdate = '女'
SET @xUpdate.modify('
```

```
    replace value of (/Peoples/People/SEX/text())[1]
    with sql:variable("@SEXUpdate")' )
SELECT @xUpdate


 --2.insert() attribute
DECLARE @xinsert XML
SELECT @xinsert ='<People NAME="dongsheng" />'
DECLARE @SEX VARCHAR(15)
SELECT @SEX = '男'
SET @xinsert.modify('    insert attribute SEX {sql:variable("@SEX")} as
last into
(/People)[1]')
SELECT @xinsert

--2.insert() element
DECLARE @xinsert1 XML
SELECT @xinsert1 ='<People NAME="dongsheng" />'
DECLARE @SEX1 VARCHAR(15)
SELECT @SEX1 = '男'
SET @xinsert1.modify('    insert element SEX {sql:variable("@SEX1")} as
last into
(/People)[1]')
SELECT @xinsert1

--3.delete  Attribute
DECLARE @xdelete XML
SELECT @xdelete = '<People NAME="dongsheng" SEX="男"/>'
SET @xdelete.modify('    delete (/People/@SEX)[1] ')
SELECT @xdelete

--3 delete  Element
DECLARE @xdelete1 XML
SELECT @xdelete1 = '
<Peoples>
  <People>
      <NAME>土豆</NAME>
      <SEX>男</SEX>
      <QQ>5345454554</QQ>
  </People>
</Peoples>'
```

```
SELECT @xdelete1
SET @xdelete1.modify('
    delete (/Peoples/People/SEX)[1]'
 )
SELECT @xdelete1
```

如使用 FOR XML 构造 XML 中所述，PATH 模式提供了一种较简单的方法来混合元素和属性。PATH 模式还是一种用于引入附加嵌套来表示复杂属性的较简单的方法。尽管您可以使用 FOR XML EXPLICIT 模式查询从行集构造这种 XML，但 PATH 模式为可能很麻烦的 EXPLICIT 模式查询提供了一种较简单的替代方法。通过 PATH 模式，以及用于编写嵌套 FOR XML 查询的功能和返回 xml 类型实例的 TYPE 指令，您可以编写简单的查询。

在 PATH 模式中，列名或列别名被作为 XPath 表达式来处理。这些表达式指明了如何将值映射到 XML。每个 XPath 表达式都是一个相对 XPath，它提供了项类型（例如属性、元素和标量值）以及将相对于行元素而生成的节点的名称和层次结构。

没有名称的列

具有名称的列

名称被指定为通配符 (*) 的列

以某个 XPath 节点测试的名称命名的列

以指定为 data() 的路径命名的列

默认情况下包含 NULL 值的列

```
/*
经常在论坛看到高手使用了for xml path，由于是搜索一下，记录了详细的使用方法。
在SQL Server中利用FOR XML PATH 语句能够把查询的数据生成XML数据，
下面是它的一些应用示例。
*/
DECLARE @TempTable table(UserID int , UserName nvarchar(50));
insert into @TempTable (UserID,UserName) values (1,'a')
insert into @TempTable (UserID,UserName) values (2,'b')
```

```
select UserID,UserName from @TempTable FOR XML PATH
/*运行这段脚本，将生成如下结果：
<row>
  <UserID>1</UserID>
  <UserName>a</UserName>
</row>
<row>
  <UserID>2</UserID>
  <UserName>b</UserName>
</row> */
--大家可以看到两行数据生成了两个节点，修改一下PATH的参数：
select UserID,UserName from @TempTable FOR XML PATH('lzy')
/*再次运行上述脚本，将生成如下的结果：
<lzy>
  <UserID>1</UserID>
  <UserName>a</UserName>
</lzy>
<lzy>
  <UserID>2</UserID>
  <UserName>b</UserName>
</lzy>
可以看到节点变成，其实PATH() 括号内的参数是控制节点名称的，
这样的话大家可以看一下如果是空字符串（不是没有参数）会是什么结果?*/
select UserID,UserName from @TempTable FOR XML PATH('')
--执行上面这段脚本将生成结果：
/*
<UserID>1</UserID>
<UserName>a</UserName>
<UserID>2</UserID>
<UserName>b</UserName>
这样就不显示上级节点了，大家知道在PATH 模式中，
列名或列别名被作为XPath 表达式来处理，也就是说，是列的名字，
这样大胆试验一下不给指定列名和别名会是怎么样？*/
select CAST(UserID AS varchar) + '',UserName + '' from @TempTable FOR XML
PATH('')
/*所有数据都生成一行，而且还没有连接字符，这样的数据可能对大家没有用处，还可以再变
化一下：*/
select CAST(UserID AS varchar) + ',',UserName + '',';' from @TempTable
FOR XML PATH('')
/*生成结果
1,a;2,b;
```

```
大家现在明白了吧，可以通过控制参数来生成自己想要的结果，例如：*/
select '{' + CAST(UserID AS varchar) + ','''' +UserName + '''','}' from
@TempTable FOR XML PATH('')

/*生成结果
{1,"a"}{2,"b"}
还可以生成其他格式，大家可以根据自己需要的格式进行组合。
下面是一个数据统计的应用，希望大家可以通过下面的实例想到更多的应用
*/
DECLARE @T1 table(UserID int , UserName nvarchar(50),CityName
nvarchar(50));
insert into @T1 (UserID,UserName,CityName) values (1,'a','上海')
insert into @T1 (UserID,UserName,CityName) values (2,'b','北京')
insert into @T1 (UserID,UserName,CityName) values (3,'c','上海')
insert into @T1 (UserID,UserName,CityName) values (4,'d','北京')
insert into @T1 (UserID,UserName,CityName) values (5,'e','上海')

select CityName,Stuff((select ','+UserName from @T1 where
cityName=a.CityName for xml path(''))
              ,1,1,'') as userName from @T1 a
group by CityName


SELECT B.CityName,LEFT(UserList,LEN(UserList)-1) FROM (
SELECT CityName,
  (SELECT UserName+',' FROM @T1 WHERE CityName=A.CityName FOR XML PATH(''))
AS UserList
FROM @T1 A
GROUP BY CityName
) B
```

outer apply

用outer apply最后最外层的select是获取外部表表达式结果集和表值函数的结果集，是两个结果集，而outer apply两边各是一个结果集，只是表值函数的结果集中可以根据其结果集别名获取并调用外部表表达式的每行，这个过程反过来不行

数据库项目文档

```
/*
作者：梁勇
时间：-12-9
name value
---- ---------
张三  1,2,3,4
李四  a,s,d,f,g
转为
name valueb
---- -----------
张三  1
张三  2
张三  3
张三  4
李四  a
李四  s
李四  d
李四  f
李四  g
*/
use City;
go
Set nocount on
if object_id('tb') is not null
drop table tb
go
create table tb([name] nvarchar(4),[value] nvarchar(9))
insert tb
select '张三','1,2,3,4' union all
select '李四','a,s,d,f,g'
go
SELECT
a.[name],b.[valueb]
FROM (SELECT
[name],[value]=CAST('<v>'+REPLACE([value],',','</v><v>')+'</v>' AS xml)
FROM tb) a
OUTER APPLY (
SELECT [valueb]=C.value('.','varchar(50)')
FROM
a.[value].nodes('/v') AS T(C)
```

```
) b
```

Cross apply 与 Outer apply 区别

```
/*
透过执行计划可以看出，cross apply类似不带where条件的连接即cross join 。
形式上会灵活些.
使用APPLY 运算符可以为实现查询操作的外部表表达式返回的每个行调用表值函数。
表值函数作为右输入，外部表表达式作为左输入
。通过对右输入求值来获得左输入每一行的计算结果，生成的行被组合起来作为最终输出。
APPLY 运算符生成的列的列表是左输入中
的列集，后跟右输入返回的列的列表。
APPLY 有两种形式：CROSS APPLY 和OUTER APPLY。CROSS APPLY 仅返回外部表中通过
表值函数生成结果集的行。OUTER APPLY 既返
回生成结果集的行，也返回不生成结果集的行，其中表值函数生成的列中的值为NULL。
*/
DECLARE @t table (姓名varchar(10))
insert into @T values('张三')
insert into @T values('李四')
insert into @T values('王二')
insert into @T values(NULL )
DECLARE @T2 table (姓名varchar(10) , 课程varchar(10) , 分数int)
insert into @T2 values('张三' , '语文' , 74)
insert into @T2 values('张三' , '数学' , 83)
insert into @T2 values('张三' , '物理' , 93)
insert into @T2 values('李四' , '物理' , 91)
insert into @T2 values(NULL , '数学' , 50)
--select * from @T
--select * from @T2
select * from @T a
cross apply
    (select 课程,分数from @t2 where 姓名=a.姓名) b

        select
    *
from
    @T a
outer apply
    (select 课程,分数from @t2 where 姓名=a.姓名) b
```

数据库项目文档

```
/*
 我们知道有个SQL Server 2000 中有个cross join 是用于交叉联接的。实际上增加
 cross apply 和outer apply 是用于交叉联接表值函数（返回表结果集的函数）的，
  更重要的是这个函数的参数是另一个表中的字段。这个解释可能有些含混不请，请看下面的例子：
*/
-- 1. cross join 联接两个表
select *
  from TABLE_1 as T1
 cross join TABLE_2 as T2
-- 2. cross join 联接表和表值函数，表值函数的参数是个“常量”
select *
  from TABLE_1 T1
 cross join FN_TableValue(100)
-- 3. cross join  联接表和表值函数，表值函数的参数是“表T1中的字段”
select *
  from TABLE_1 T1
 cross join FN_TableValue(T1.column_a)
/*
Msg 4104, Level 16, State 1, Line 1
The multi-part identifier "T1.column_a" could not be bound.
最后的这个查询的语法有错误。在cross join 时，表值函数的参数不能是表T1 的字段，为啥不能这样做呢？我猜可能微软当时没有加这个功能：），后来有客户抱怨后，于是微软就增加了cross apply 和outer apply 来完善，请看cross apply, outer apply 的例子：
*/
-- 4. cross apply
select *
  from TABLE_1 T1
 cross apply FN_TableValue(T1.column_a)

-- 5. outer apply
select *
  from TABLE_1 T1
 outer apply FN_TableValue(T1.column_a)

/*
cross apply 和outer apply 对于T1 中的每一行都和派生表（表值函数根据T1当前行数据生成的动态结果集）做了一个交叉联接。cross apply 和outer apply 的区别在于：如果根据T1 的某行数据生成的派生表为空，cross apply 后的结果集就不包含T1 中的这行数
```

据，而outer apply 仍会包含这行数据，并且派生表的所有字段值都为NULL。

下面的例子摘自微软SQL Server 2005 联机帮助，它很清楚的展现了cross apply 和outer apply 的不同之处：

```
*/
-- cross apply
select *
  from Departments as D
 cross apply fn_getsubtree(D.deptmgrid) as ST
--deptid     deptname     deptmgrid   empid       empname       mgrid
lvl
------------ -----------   ----------- ----------- -----------
----------- ------
--1          HR           2           2           Andrew        1           0
--1          HR           2           5           Steven        2           1
--1          HR           2           6           Michael       2           1
--2          Marketing     7           7           Robert        3
0
--2          Marketing     7           11          David         7
1
--2          Marketing     7           12          Ron           7           1
--2          Marketing     7           13          Dan           7           1
--2          Marketing     7           14          James         11
2
--3          Finance       8           8           Laura         3           0
--4          R&D           9           9           Ann           3           0
--5          Training      4           4           Margaret      1
0
--5          Training      4           10          Ina           4           1

--(12 row(s) affected)
-- outer apply
select *
  from Departments as D
 outer apply fn_getsubtree(D.deptmgrid) as ST
 /*
deptid     deptname     deptmgrid   empid       empname       mgrid
lvl
----------- -----------   ----------- ----------- -----------
----------- ------
1          HR           2           2           Andrew        1           0
```

```
1           HR           2           5           Steven       2           1
1           HR           2           6           Michael      2           1
2           Marketing    7           7           Robert       3           0
2           Marketing    7           11          David        7           1
2           Marketing    7           12          Ron          7           1
2           Marketing    7           13          Dan          7           1
2           Marketing    7           14          James        11          2
3           Finance      8           8           Laura        3           0
4           R&D          9           9           Ann          3           0
5           Training     4           4           Margaret     1           0
5           Training     4           10          Ina          4           1
6           Gardening    NULL        NULL        NULL         NULL
NULL

(13 row(s) affected)
```

注意outer apply 结果集中多出的最后一行。当Departments 的最后一行在进行交叉联接时：

deptmgrid 为NULL，fn_getsubtree(D.deptmgrid) 生成的派生表中没有数据，但outer apply

仍会包含这一行数据，这就是它和cross join 的不同之处。

下面是完整的测试代码，你可以在SQL Server 2005 联机帮助上找到：

```
*/
-- create Employees table and insert values
IF OBJECT_ID('Employees') IS NOT NULL
 DROP TABLE Employees
GO
CREATE TABLE Employees
(
 empid INT NOT NULL,
 mgrid INT NULL,
 empname VARCHAR(25) NOT NULL,
 salary MONEY NOT NULL
)
GO
IF OBJECT_ID('Departments') IS NOT NULL
 DROP TABLE Departments
GO
-- create Departments table and insert values
CREATE TABLE Departments
(
```

```
 deptid INT NOT NULL PRIMARY KEY,
 deptname VARCHAR(25) NOT NULL,
 deptmgrid INT
)
GO

-- fill datas
INSERT  INTO employees VALUES  (1,NULL,'Nancy',00.00)
INSERT  INTO employees VALUES  (2,1,'Andrew',00.00)
INSERT  INTO employees VALUES  (3,1,'Janet',00.00)
INSERT  INTO employees VALUES  (4,1,'Margaret',00.00)
INSERT  INTO employees VALUES  (5,2,'Steven',00.00)
INSERT  INTO employees VALUES  (6,2,'Michael',00.00)
INSERT  INTO employees VALUES  (7,3,'Robert',00.00)
INSERT  INTO employees VALUES  (8,3,'Laura',00.00)
INSERT  INTO employees VALUES  (9,3,'Ann',00.00)
INSERT  INTO employees VALUES  (10,4,'Ina',00.00)
INSERT  INTO employees VALUES  (11,7,'David',00.00)
INSERT  INTO employees VALUES  (12,7,'Ron',00.00)
INSERT  INTO employees VALUES  (13,7,'Dan',00.00)
INSERT  INTO employees VALUES  (14,11,'James',00.00)

INSERT  INTO departments VALUES  (1,'HR',2)
INSERT  INTO departments VALUES  (2,'Marketing',7)
INSERT  INTO departments VALUES  (3,'Finance',8)
INSERT  INTO departments VALUES  (4,'R&D',9)
INSERT  INTO departments VALUES  (5,'Training',4)
INSERT  INTO departments VALUES  (6,'Gardening',NULL)
GO
--SELECT * FROM departments

-- table-value function
IF OBJECT_ID('fn_getsubtree') IS NOT NULL
 DROP FUNCTION  fn_getsubtree
GO
CREATE  FUNCTION dbo.fn_getsubtree(@empid AS INT)
RETURNS TABLE
AS
RETURN(
  WITH Employees_Subtree(empid, empname, mgrid, lvl)
  AS
```

```
  (
    -- Anchor Member (AM)
    SELECT empid, empname, mgrid, 0
    FROM employees
    WHERE empid = @empid
    UNION ALL
    -- Recursive Member (RM)
    SELECT e.empid, e.empname, e.mgrid, es.lvl+1
    FROM employees AS e
       join employees_subtree AS es
          ON e.mgrid = es.empid
  )
    SELECT * FROM Employees_Subtree
)
GO

-- cross apply query
SELECT  *
FROM Departments AS D
    CROSS APPLY fn_getsubtree(D.deptmgrid) AS ST



-- outer apply query
SELECT  *
FROM Departments AS D
    OUTER APPLY fn_getsubtree(D.deptmgrid) AS ST



DECLARE @A TABLE(
    id int)
INSERT @A
SELECT id = 1 UNION ALL
SELECT id = 2

DECLARE @B TABLE(
    id int)
INSERT @B
SELECT id = 1 UNION ALL
SELECT id = 3
```

```
-- 1. 右输入为表时, APPLY操作符与CROSS JOIN的结果一样
SELECT *
FROM @A
    CROSS APPLY @B

-- 2. 右输入为派生表时, 可以用APPLY操作符模拟JOIN
-- 2.a 模拟INNER JOIN
SELECT *
FROM @A A
    CROSS APPLY(
        SELECT * FROM @B
        WHERE id = A.id
    )B

-- 2.b 模拟LEFT JOIN
SELECT *
FROM @A A
    OUTER APPLY(
        SELECT * FROM @B
        WHERE id = A.id
    )B
/*
(2 行受影响)

(2 行受影响)
id          id
----------- -----------
1           1
2           1
1           3
2           3

(4 行受影响)

id          id
----------- -----------
1           1

(1 行受影响)
```

```
id          id
----------- -----------
1           1
2           NULL

(2 行受影响)

*/


USE tempdb
GO

-- 用于分拆字符串的表值函数
CREATE FUNCTION dbo.f_Split(
    @str varchar(max)
)RETURNS @re TABLE(
       id int IDENTITY, val varchar(10))
AS
BEGIN
    DECLARE @pos int
    SET @pos = CHARINDEX(',', @str)
    WHILE @pos > 0
    BEGIN
        INSERT @re(val) VALUES(LEFT(@str, @pos - 1))
        SELECT
            @str = STUFF(@str, 1, @pos, ''),
            @pos = CHARINDEX(',', @str)
    END
    IF @str > ''
        INSERT @re(val) VALUES(@str)
    RETURN
END
GO

-- 用于分拆的示例数据
DECLARE @tb TABLE(
    col varchar(max))
INSERT @tb
SELECT col = '1,2,3' UNION ALL
SELECT col = NULL UNION ALL
```

```
SELECT col = '' UNION ALL
SELECT col = '1'

-- 使用CROSS APPLY
SELECT *
FROM @tb A
    CROSS APPLY dbo.f_Split(A.col) B

-- 使用OUTER APPLY
SELECT *
FROM @tb A
    OUTER APPLY dbo.f_Split(A.col) B
GO

-- 删除演示数据
DROP FUNCTION dbo.f_Split
/*
(4 行受影响)
col
id          val
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
-------------------------------------------------- -----------
----------
1,2,3
1           1
1,2,3
2           2
1,2,3
3           3
1
1           1

(4 行受影响)

col
id          val
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
```

数据库项目文档

```
---------------------------------------------- -----------
----------
1,2,3
1           1
1,2,3
2           2
1,2,3
3           3
NULL
NULL        NULL

NULL        NULL
1
1           1

(6 行受影响)

*/

create table #T(姓名varchar(10))
insert into #T values('张三')
insert into #T values('李四')



create table #T2(姓名varchar(10) , 课程varchar(10) , 分数int)
insert into #T2 values('张三' , '语文' , 74)
insert into #T2 values('张三' , '数学' , 83)
insert into #T2 values('张三' , '物理' , 93)
insert into #T2 values('李四' , '语文' , 90)
insert into #T2 values('李四' , '数学' , 66)
insert into #T2 values('李四' , '物理' , 88)



-- cross apply的经典用法;求每个人考试科目中成绩的前两名(取出每组中的前几名)
select * from #t t
 cross apply
(select top 2 * from #t2 where 姓名=t.姓名order by 分数desc) as o

/*
```

```
姓名         姓名         课程         分数
---------- ---------- ---------- -----------
张三         张三         物理         93
张三         张三         数学         83
李四         李四         语文         90
李四         李四         物理         88
*/
```

Pivot 写一个九九乘法表

```
/*
作者：梁勇
时间：-12-24
描述：用pivot表示九九乘法表
引申：可以试想很多结果都可以通过pivot来实现的。假设一个结果有很多列，且每个列看起来都是一样的规律，那么可否将这些列合并到一个列(而这样对于sql代码来说很容易实现)。然后利用pivot将一列转为多列,继而得到用户的效果,如九九乘法表或者日历,因为每列都有规律，那么假设他们在一列上这是很好实现的，接下来用pivot进行转换就可以了，估计日历也是这样的。
重点：一般是先只有两列，一定要外加一列，该列是用来分组的，该列的值就是使得最后结果中的每一行都只分别对应一个值，即每行中所有单元格的数据都对应这个值，而不同的行对应不同的值，比如下面的RR，以及下个例子日历的SS，他们的功能都是一样的，就是要使结果分组（即作为透视分组列）
*/
with cte1 as
(
  select top 9 ROW_NUMBER()over(order by getdate())as RN from sysobjects
)
,cte2 as
(
select A.RN,B.Va,ROW_NUMBER()over(PARTITION by RN order by getdate())as
RR from
(select * from cte1)as A outer apply(select case when A.RN<=RN then
(ltrim(A.RN)+'X'+ltrim(RN)+'='+LTRIM(A.RN*RN))
                                          else '' end as Va from cte1 )
as B
)
select * from cte2 a
```

```
pivot
(
   max(Va)
   for RN in([1],[2],[3],[4],[5],[6],[7],[8],[9])
)as B
```

Pivot 写一个日历

```
/*
作者：梁勇
时间：-12-25
描述：用pivot来实现日历

datepart(yy，时间)[yy]是指右边[时间]中包含的[yy]的数据，是个整数,如：右边是2011-12-25，那么就返回年份
datediff(返回格式单位,starttime,endTime) 最后是一个整型数据，单位是返回格式单位，如：右边是datediff(mm,'-11-24',2010-12-25),返回的是两个日期之前的总月份，为
dateadd(addCount的单位，addCount,BaseTime)，addCount添加数量的单位，最后是一个时间格式，如：dateadd(mm,3,'2010-8-24'),表示在原来的基础上添加个月，最后显示2010-11-24
实例：
1、获取给定时间的当月的第一天：先获取给定时间的总月份，然后给一个以为起始日期的时间添加这么多月份，就是了
2、获取给定时间的当月的上个月的最后一天：就是将给定时间的当月下一个月的第一天减去3ms即可
3、获取给定日期时当个月的天数，就求这个月的第一天和下个月的第一天差几天即可
*/
set nocount on
declare @thetime datetime
set @thetime='2022-2-25'
;
with cte1 as
(select * from
   (select top 31 ROW_NUMBER()over(order by getdate())as RN from sysobjects) as A,
   (select datediff(day,dateadd(mm,DATEDIFF(MM,0,@thetime),0),dateadd(mm,DATEDIFF(MM,0,@thetime)+1,0)) as dd)As B
    where B.dd>=A.RN
```

```
)
,t2 as
(
select RN,

xq=substring(datename(WEEKDAY,DATEADD(DAY,RN-1,dateadd(mm,DATEDIFF(MM
,0,@thetime),0))),3,1),

ss=datepart(Week,DATEADD(DAY,RN-1,dateadd(mm,DATEDIFF(MM,0,@thetime),
0)))
      from cte1
)
select ss,isnull(日,'')as 日,isnull(一,'')as 一,isnull(二,'')as
三,isnull(三,'')as 三,isnull(四,'')as 四,isnull(五,'')as
五,isnull(六,'')as 六 from t2
pivot
(
  max(RN)
  for xq in([日],[一],[二],[三],[四],[五],[六])
)as B
```

sql except 和 INTERSECT 用法

比较两个查询的结果，返回非重复值。

EXCEPT 从左查询中返回右查询没有找到的所有非重复值。

INTERSECT 返回 INTERSECT 操作数左右两边的两个查询都返回的所有非重复值。

以下是将使用 EXCEPT 或 INTERSECT 的两个查询的结果集组合起来的基本规则：

所有查询中的列数和列的顺序必须相同。

数据类型必须兼容。

Transact-SQL 语法约定

语法

```
{ <query_specification> | ( <query_expression> ) }
{ EXCEPT | INTERSECT }
{ <query_specification> | ( <query_expression> ) }
```

参数

<query_specification> | (<query_expression>)

查询规范或查询表达式返回与来自另一个查询规范或查询表达式的数据相比较的数据。在 EXCEPT 或 INTERSECT 运算中，列的定义可以不同，但它们必须在隐式转换后进行比较。如果数据类型不同，则用于执行比较并返回结果的类型是基于数据类型优先级的规则确定的。

如果类型相同，但精度、小数位数或长度不同，则根据用于合并表达式的相同规则来确定结果。有关详细信息，请参阅 精度、小数位数和长度 (Transact-SQL)。

查询规范或表达式不能返回 xml、text、ntext、image 或非二进制 CLR 用户定义类型列，因为这些数据类型不可比较。

EXCEPT

从 EXCEPT 操作数左边的查询中返回右边的查询未返回的所有非重复值。

INTERSECT

返回 INTERSECT 操作数左右两边的两个查询均返回的所有非重复值。

备注

如果 EXCEPT 或 INTERSECT 操作数左边和右边的查询返回的可比较列的数据类型是具有不同排序规则的字符数据类型，则根据排序规则优先级的规则执行所需的比较。如果无法执行此转换，Microsoft SQL Server 2005 数据库引擎将返回错误。

通过比较行来确定非重复值时，两个 NULL 值被视为相等。

EXCEPT 或 INTERSECT 返回的结果集的列名与操作数左侧的查询返回的列名相同。

ORDER BY 子句中的列名或别名必须引用左侧查询返回的列名。

EXCEPT 或 INTERSECT 返回的结果集中的任何列的为空性与操作数左侧的查询返回的对应列的为空性相同。

如果 EXCEPT 或 INTERSECT 与表达式中的其他运算符一起使用，则在以下优先顺序的上下文中对其进行评估：

括号中的表达式

INTERSECT 操作数

基于在表达式中的位置从左到右求值的 EXCEPT 和 UNION

如果 EXCEPT 或 INTERSECT 用于比较两个以上的查询集，则数据类型转换是通过一次比较两个查询来确定的，并遵循前面提到的表达式求值规则。

EXCEPT 和 INTERSECT 不能在分布式分区视图定义、查询通知中使用，也不能与 COMPUTE 和 COMPUTE BY 子句一起使用。

EXCEPT 和 INTERSECT 可在分布式查询中使用，但只在本地服务器上执行，不会被推送到链接服务器。因此，在分布式查询中使用 EXCEPT 和 INTERSECT 可能会影响性能。

快速只进游标和静态游标与 EXCEPT 或 INTERSECT 运算一起使用时，在结果集中完全受支持。如果由键集驱动的游标或动态游标与 EXCEPT 或 INTERSECT 运算一起使用，则运算的结果集的游标转换为静态游标。

使用 SQL Server Management Studio 中的图形显示计划功能显示 EXCEPT 运算时，该运算显示为 left anti semi join，INTERSECT 运算显示为 left semi join。

例：select fld7 from table4 except select fld7 from table5

3.SQL 查询的技巧，行转列，列装行。

结果集转化为一列(When Case)

```
/*
标题：普通行列转换(version 2.0)
```

数据库项目文档

作者：爱新觉罗.毓华(十八年风雨,守得冰山雪莲花开)
时间：2008-03-09
地点：广东深圳
说明：普通行列转换(version 1.0)仅针对sql server 2000提供静态和动态写法，version 2.0增加sql server 2005的有关写法。

问题：假设有张学生成绩表(tb)如下：
姓名 课程 分数
张三 语文 74
张三 数学 83
张三 物理 93
李四 语文 74
李四 数学 84
李四 物理 94
想变成(得到如下结果)：
姓名 语文 数学 物理
---- ---- ---- ----
李四 74 84 94
张三 74 83 93

```
*/

create table tb(姓名 varchar(10) , 课程 varchar(10) , 分数 int)
insert into tb values('张三' , '语文' , 74)
insert into tb values('张三' , '数学' , 83)
insert into tb values('张三' , '物理' , 93)
insert into tb values('李四' , '语文' , 74)
insert into tb values('李四' , '数学' , 84)
insert into tb values('李四' , '物理' , 94)
go

--SQL SERVER 2000 静态SQL,指课程只有语文、数学、物理这三门课程。(以下同)
select 姓名 as 姓名 ,
  max(case 课程 when '语文' then 分数 else 0 end) 语文,
  max(case 课程 when '数学' then 分数 else 0 end) 数学,
  max(case 课程 when '物理' then 分数 else 0 end) 物理
from tb
group by 姓名

--SQL SERVER 2000 动态SQL,指课程不止语文、数学、物理这三门课程。(以下同)
declare @sql varchar(8000)
```

```
set @sql = 'select 姓名 '
select @sql = @sql + ' , max(case 课程 when ''' + 课程 + ''' then 分数 else
0 end) [' + 课程 + ']'
from (select distinct 课程 from tb) as a
set @sql = @sql + ' from tb group by 姓名'
exec(@sql)

--SQL SERVER 2005 静态SQL。
select * from (select * from tb) a pivot (max(分数) for 课程 in (语文,数
学,物理)) b

--SQL SERVER 2005 动态SQL。
declare @sql varchar(8000)
select @sql = isnull(@sql + '],[' , '') + 课程 from tb group by 课程
set @sql = '[' + @sql + ']'
exec ('select * from (select * from tb) a pivot (max(分数) for 课程 in ('
+ @sql + ')) b')

---------------------------------

/*
问题：在上述结果的基础上加平均分，总分，得到如下结果：
姓名 语文 数学 物理 平均分 总分
---- ---- ---- ---- ------ ----
李四 74   84   94   84.00  252
张三 74   83   93   83.33  250
*/

--SQL SERVER 2000 静态SQL。
select 姓名 姓名,
  max(case 课程 when '语文' then 分数 else 0 end) 语文,
  max(case 课程 when '数学' then 分数 else 0 end) 数学,
  max(case 课程 when '物理' then 分数 else 0 end) 物理,
  cast(avg(分数*1.0) as decimal(18,2)) 平均分,
  sum(分数) 总分
from tb
group by 姓名

--SQL SERVER 2000 动态SQL。
declare @sql varchar(8000)
set @sql = 'select 姓名 '
```

```
select @sql = @sql + ' , max(case 课程 when ''' + 课程 + ''' then 分数 else 0 end) [' + 课程 + ']'
from (select distinct 课程 from tb) as a
set @sql = @sql + ' , cast(avg(分数*1.0) as decimal(18,2)) 平均分 , sum(分数) 总分 from tb group by 姓名'
exec(@sql)

--SQL SERVER 2005 静态SQL。
select m.* , n.平均分 , n.总分 from
(select * from (select * from tb) a pivot (max(分数) for 课程 in (语文,数学,物理)) b) m,
(select 姓名 , cast(avg(分数*1.0) as decimal(18,2)) 平均分 , sum(分数) 总分 from tb group by 姓名) n
where m.姓名 = n.姓名

--SQL SERVER 2005 动态SQL。
declare @sql varchar(8000)
select @sql = isnull(@sql + ',' , '') + 课程 from tb group by 课程
exec ('select m.* , n.平均分 , n.总分 from
(select * from (select * from tb) a pivot (max(分数) for 课程 in (' + @sql + ')) b) m ,
(select 姓名 , cast(avg(分数*1.0) as decimal(18,2)) 平均分 , sum(分数) 总分 from tb group by 姓名) n
where m.姓名 = n.姓名')

drop table tb

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

结果集转化为多列(使用窗口函数以及 Case 子句进行配合)

```
/*
用SQL实现结果集的各种转换----结果集转成多列.
*/
if object_id(N'CTEZhuan','U') is not null drop table CTEZhuan
create table CTEZhuan(Id int,ProvinceName varchar(50) , CityName varchar(20) , CityPop int)
```

```
go
insert into CTEZhuan values(26,'福建' , '南平' , 20)
insert into CTEZhuan values(27,'福建' , '三明' , 10)
insert into CTEZhuan values(28,'福建' , '泉州' , 10)


insert into CTEZhuan values(26,'湖南' , '长沙' , 20)
insert into CTEZhuan values(27,'湖南' , '常德' , 20)

insert into CTEZhuan values(28,'浙江' , '嘉兴' , 20)


;with cte as
(
select *,ROW_NUMBER() OVER(PARTITION BY ProvinceName ORDER BY CityName)
AS RN
from CTEZhuan
)
--select * from cte

select
       max(Case when ProvinceName='福建' then CityName else '' end) as '
福建',
       max(Case when ProvinceName='湖南' then CityName else '' end) as
'湖南',
       max(Case when ProvinceName='浙江' then CityName else '' end) as
'浙江'
 from cte group by RN
/*
 接下来讲解下上面这段SQL语句。

 1~6用了公用表表达式CTE(优点:、没有像临时表那样耗性能，也没有像子查询那样可读性差。
2、增强了可维护性和效率)定义了一个临时数据集(暂且这么描述)，然后在第行中引用这个公
用表数据集，
关于CTE大家可以到其他网站再好好了解，这里不再详细说明。
同时上面的SQL也涉及到了SQL的窗口函数row_number() over(partition by
ProvinceName order By CityName).

row_number() over (order by  ColumnName)函数是SQL2005
在SQL2000的基础上新增的一个函数，作用是根据ColumnName这个列进行排序并编号，
编号从开始。
```

row_number() over (order by ColumnName)函数是SQL2005
在SQL2000的基础上新增的一个函数，作用是根据ColumnName这个列进行排序并编号，
编号从开始。

举个例子：

```
select *,
row_number() over( order by CityName)
as RN from CityTable
```

执行上面的SQL语句，将结果集按照CityName这个列进行排序，并且编号，
编号从开始，将得到如图所示的结果集

而partition by ProvinceName是将整个结果集先根据ProvinceName进行分区
(这里是把每个省划分为一个分区)，再对各个不同的分区根据CityName进行排序，并重新编号，
编号从开始。

```
select *,
row_number() over(partition by ProvinceName order by CityName)
as RN from CityTable
*/
```

结果集转化为多行(使用 Case 之句和笛卡尔积)

```
/*
用SQL实现结果集的各种转换----结果集转成多行
 关于将结果集转换成列，我们使用窗口函数以及Case子句进行配合，
而将结果集转成一行或者多行，则要使用到Case子句和笛卡尔积。

 .
*/
if object_id(N'TableA','U') is not null drop table TableA
create table TableA(ProvinceName varchar(50))
go

if object_id(N'TableB','U') is not null drop table TableB
create table TableB(福建varchar(50),湖南varchar(50),浙江varchar(50))
go
```

```
insert into TableA values('福建')
insert into TableA values('湖南')
insert into TableA values('浙江')
go

insert into TableB values('南平','长沙','嘉兴')
insert into TableB values('泉州','常德','')
insert into TableB values('三明','','')
go

with cte as (
select TableB.*, TableA.*from TableA ,
 TableB
),cte2 as (

select
--*  from cte
ProvinceName,
case

    when ProvinceName='福建' then 福建
    when ProvinceName='湖南' then 湖南
    when ProvinceName='浙江' then 浙江
    else ''
    end as CityName
 from cte
)
select * from cte2 where cityName <> ''
group by ProvinceName,CityName
```

结果集转化为一列(C ase 之句和笛卡尔积)

```
/*
之前的四篇关于用SQL将结果集进行转换的知识点，个人感觉可能就是行转为列时有点伤脑筋，但是找到方法后也还是可以解决的，其中有一点，对于每组中的第N行假设要一起排在结果集中的同一行,那么，那么就设法让每组中的第N行的某个列(假设为RN)的值为相同，其中用窗口函数Row_Number()方法可以实现,这样只要按照RN分组外加聚合函数(如Max)即可实现,然后在
Case中判断RN列的值。而对于列转行则直接使用笛卡尔积，使得原表中的每一行(假设有M列)都
```

数据库项目文档

具有M行，接下来要让这M行可以根据某两个列(其中一个进行分区，另外一个进行
Row_Number()
over(order by 列名) as RN)，使得每行具有不同的一个RN值，这样就可以根据Case子句进行最后的组合。

一般情况下我们只需要利用窗口函数(row_number() over (partition by column1 order by column2))
和Case子句进行行转列，而利用笛卡尔积和Case子句进行列转行.

下面介绍一种很特殊的情况，将所有的行转换为一列，将图所示的结果集转换为图所示的结果集

```
*/
if object_id(N'CTEZhuan','U') is not null drop table CTEZhuan
create table CTEZhuan(Id int,ProvinceName varchar(50) , CityName
varchar(20) , CityPop int)
go
insert into CTEZhuan values(26,'福建' , '南平' , 20)
insert into CTEZhuan values(28,'福建' , '泉州' , 10)
insert into CTEZhuan values(26,'湖南' , '长沙' , 20)
/*
根据前几章的介绍，我们这里要将图的结果集转换为图的结果集，
其实就是将行转为列。这种特殊的行转列，我们用笛卡尔积和case子句即可解决。
首先将图中的结果集进行笛卡尔运算，使得每行都变为相应的三行。如下SQL语句：
*/
select ProvinceName,CityName,CityPop from CTEZhuan,
(select top 3 ID from CTEZhuan) as Table1

/*
现在我们要将每个ProvinceName进行分区，将每个
ProvinceName里的所有CityName进行排序，如下SQL语句
*/
;with TableOne as
(
select ProvinceName,CityName,CityPop from CTEZhuan,
(select top 3 ID from CTEZhuan) as Table1
)
select *,row_number() over(partition by CityName order by CityName) as
RN from TableOne

/*接下来根据不同的RN值利用Case子句进行判断，即可得到结果*/
```

数据库项目文档

```
;with TableOne as
(
select ProvinceName,CityName,CityPop from CTEZhuan,
(select top 3 ID from CTEZhuan) as Table1
)
,TableTwo as
(
select *,row_number() over(partition by CityName order by CityName) as
RN from TableOne
)
select
case when RN=1 then ProvinceName
     when RN=2 then CityName
     when RN=3 then cast(CityPop as char(10))
     else ''
     end as '结果'
from TableTwo
order by ProvinceName

/*  该过程中遇到一些问题，如下：

 一、错误信息：除非另外还指定了TOP 或FOR XML，否则，
ORDER BY 子句在视图、内联函数、派生表、子查询和公用表表达式中无效。

SQL语句如下：

*/

;with TableOne as
(
select ProvinceName,CityName,CityPop from CTEZhuan,
(select top 3 ID from CTEZhuan) as Table1
)
,TableTwo as
(
select *,row_number() over(partition by CityName order by CityName) as
RN from TableOne
order by ProvinceName
)
select
case when RN=1 then ProvinceName
```

```
    when RN=2 then CityName
     when RN=3 then cast(CityPop as char(10))
    else ''
     end as '结果'
from TableTwo
--order by ProvinceName
/*
将第行的order by ProvinceName放在了第行，出错的原因：针对一个表的SELECT
其实并不是返回一个表，而是一个游标；作为子查询，如果有order.....，规定必须
这个子查询中含有一个top关键字。于是解决办法由两种：

 第一：在该子查询中添加一个top关键字的语句:top 100 percent,即将第行改为：

select top 100 percent*,row_number() over(partition by CityName order by
CityName)
 as RN from TableOne

第二：在子查询中删除Order by 语句，将order by语句放在外部查询语句中。


*/

/*
二、下面的SQL语句中case子句里发生错误，信息如下：
在将nvarchar 值'湖南' 转换成数据类型int 时失败。
*/
;with TableOne as
(
select ProvinceName,CityName,CityPop from CTEZhuan,
(select top 3 ID from CTEZhuan) as Table1
)
,TableTwo as
(
select *,row_number() over(partition by CityName order by CityName) as
RN from TableOne
--order by ProvinceName
)
select
case when RN=1 then ProvinceName
    when RN=2 then CityName
     when RN=3 then CityPop
```

```
    else ''
     end as '结果'
from TableTwo
order by ProvinceName
/*
因为在上面的case子句中，ProvinceName和CityName是字符型的，
而CityPopulation的类型是整型，所以要将CityPopulation转换为字符型，
即将第行代码改为when RN=3 then cast(CityPopulation as char(10))即可。
*/


/*
ID          name                 value
----------- -------------------- ----------
1           张三                   aaa
2           张三                   ssss
3           张三                   ddd
4           张三                   fff
5           李四                   ggg
6           李四                   hhhh
7           李四                   jjj
8           李四                   kkk
9           李四                   ll

name                 Two
-------------------- ----------------------------

李四                   ggg,hhhh,jjj,kkk,ll
张三                   aaa,ssss,ddd,fff
*/
use City;
go
SET NOCOUNT ON
if OBJECT_ID(N'A',N'U') is not null drop table A
go
create table A
(
  ID int identity(1,1) primary key not null,
  name nvarchar(20),
  value nvarchar(10)
)
```

```
go
insert into A
select '张三', 'aaa' union all
select '张三','ssss' union all
select '张三', 'ddd' union all
select '张三', 'fff' union all
select '李四', 'ggg' union all
select '李四', 'hhhh' union all
select '李四', 'jjj' union all
select '李四', 'kkk' union all
select '李四', 'll'
go
with One as
(
select *,(value) as Re from A
)
select name,
--这里本来可以用substring函数可以实现的,但是substring函数要获取两次下面的结果集。于是用stuff，可见，在
--SQL中如要用Substring要计算两次结果集的时候就考虑用stuff。

--再者，用for xml path的是很默认生成的每个节点的名称是根据列名和列名的别名。那么不给这个选择字段列名或列名
--别名呢？那么生成的xml字符串就没有那种节点出现了。这是关键
stuff((select ','+Re from One where One.name=A.name for xml
path('')),1,1,'') as Two
from A
group by name
drop table A
--可以看出，用这种方法可以将【多行中的一列转为一行】。


/*
将上面的结果集反止：

就由原来的：

name value
---- ---------
张三  1,2,3,4
李四  a,s,d,f,g
```

转为：

```
name value
---- ------
张三  1
张三  2
张三  3
张三  4
李四  a
李四  s
李四  d
李四  f
李四  g

*/
use City;
go
Set nocount on
if object_id('tb') is not null
drop table tb
go
create table tb([name] nvarchar(4),[value] nvarchar(9))
insert tb
select '张三','1,2,3,4' union all
select '李四','a,s,d,f,g'
go
SELECT a.[name],b.[value]
FROM (SELECT
[name],[value]=CAST('<v>'+REPLACE([value],',','</v><v>')+'</v>' AS xml)
FROM tb) a
OUTER APPLY (SELECT [value]=T.C.value('.','varchar(50)') FROM
a.[value].nodes('/v') AS T(C)) b
```

结果集转化为多行转化为一列

```
/*
ID          name                value
----------- ------------------- ----------
1           张三                aaa
2           张三                ssss
3           张三                ddd
4           张三                fff
5           李四                ggg
6           李四                hhhh
7           李四                jjj
8           李四                kkk
9           李四                ll


name                Two
------------------- --------------------------

李四                ggg,hhhh,jjj,kkk,ll
张三                aaa,ssss,ddd,fff

            图
SQL语句:
*/

use City;
go
SET NOCOUNT ON
if OBJECT_ID(N'A',N'U') is not null drop table A
go
create table A
(
  ID int identity(1,1) primary key not null,
  name nvarchar(20),
 value nvarchar(10)
)
go
insert into A
select '张三', 'aaa' union all
```

```
select '张三','ssss' union all
select '张三', 'ddd' union all
select '张三', 'fff' union all
select '李四', 'ggg' union all
select '李四', 'hhhh' union all
select '李四', 'jjj' union all
select '李四', 'kkk' union all
select '李四', 'll'
go
with One as
(
select *,(value) as Re from A
)
select name,
--这里本来可以用substring函数可以实现的,但是substring函数要获取两次下面的结果集。于是用stuff，可见，在
--SQL中如要用Substring要计算两次结果集的时候就考虑用stuff。

--再者，用for xml path的是很默认生成的每个节点的名称是根据列名和列名的别名。那么不给这个选择字段列名或列名
--别名呢？那么生成的xml字符串就没有那种节点出现了。这是关键
stuff((select ','+Re from One where One.name=A.name for xml
path('')),1,1,'') as Two
from A
group by name
drop table A
--可以看出，用这种方法可以将【多行中的一列转为一行】。
use City;
go
SET NOCOUNT ON
if OBJECT_ID(N'A',N'U') is not null drop table A
go
create table A
(
  ID int identity(1,1) primary key not null,
  name nvarchar(20),
  value nvarchar(10)
)
go
insert into A
select '张三', 'aaa' union all
```

数据库项目文档

```
select '张三','ssss' union all
select '张三', 'ddd' union all
select '张三', 'fff' union all
select '李四', 'ggg' union all
select '李四', 'hhhh' union all
select '李四', 'jjj' union all
select '李四', 'kkk' union all
select '李四', 'll'
go
with One as
(
select *,(value) as Re from A
)
select name,
--这里本来可以用substring函数可以实现的,但是substring函数要获取两次下面的结果集。于是用stuff，可见，在
--SQL中如要用Substring要计算两次结果集的时候就考虑用stuff。

--再者，用for xml path的是很默认生成的每个节点的名称是根据列名和列名的别名。那么不给这个选择字段列名或列名
--别名呢？那么生成的xml字符串就没有那种节点出现了。这是关键
stuff((select ','+Re from One where One.name=A.name for xml
path('')),1,1,'') as Two
from A
group by name
drop table A
--可以看出，用这种方法可以将【多行中的一列转为一行】。



/*并且利用for xml path 可以将多行中的一列以任意效果显示！如：

name                Two
------------------- -------------------------------------

李四                 {ggg},{hhhh},{jjj},{kkk},{ll}
张三                 {aaa},{ssss},{ddd},{fff}

等等。
```

```
将上面的结果集反止：*/

use City;
go
Set nocount on
if object_id('tb') is not null
drop table tb
go
create table tb([name] nvarchar(4),[value] nvarchar(9))
insert tb
select '张三','1,2,3,4' union all
select '李四','a,s,d,f,g'
go
SELECT a.[name],b.[value]
FROM (SELECT
[name],[value]=CAST('<v>'+REPLACE([value],',','</v><v>')+'</v>' AS xml)
FROM tb) a
OUTER APPLY (SELECT [value]=T.C.value('.','varchar(50)') FROM
a.[value].nodes('/v') AS T(C)) b

/*就由原来的：

name value
---- ---------
张三  1,2,3,4
李四  a,s,d,f,g

转为：

name value
---- ------
张三  1
张三  2
张三  3
张三  4
李四  a
李四  s
李四  d
李四  f
李四  g
```

4.存储过程，函数，触发器，游标的整理

存储过程的好处和应用

(1)减少网络通信量。调用一个行数不多的存储过程与直接调用 sql 语句的网络通信量可能不会有很大的差别，可是如果存储过程包含上百行 sql 语句，那么其性能绝对比一条一条的调用 sql 语句要高得多。

(2)执行速度更快。有两个原因：首先，在存储过程创建的时候，数据库已经对其进行了一次解析和优化。其次，存储过程一旦执行，在内存中就会保留一份这个存储过程，这样下次再执行同样的存储过程时，可以从内存中直接调用。

(3)更强的适应性：由于存储过程对数据库的访问是通过存储过程来进行的，因此数据库开发人员可以在不改动存储过程接口的情况下对数据库进行任何改动，而这些改动不会对应用程序造成影响。

(4) 布式工作：应用程序和数据库的编码工作可以分别独立进行，而不会相互压制。

存储过程可以用临时表，函数不能用临时表
存储过程可以使用 Update，函数不能使用 Update
存储过程可以用 Getdate()等函数，函数不能使用 Getdate()这些函数

一、使用存储过程要注意一些问题
1、在 as 后面立即跟上一个 set nocount on 子句，这样会减少很大开销(相对于过程中存在 DML 操作的时候)
2、创建和引用过程的时候加上架构名
3、避免在返回许多行数据的 SELECT 语句中使用标量函数(getdate())。因为标量函数必须应用于每一行。
4、避免使用 SELECT *。而是应指定所需的列名称。
5、避免处理或返回"过多"的数据，一些没必要的数据就不要了
6、通过使用 BEGIN/END TRANSACTION 来使用显式事务并且保留尽可能短的事务。更长的事务意味着更长的记录锁定和更高的死锁风险
7、在 LIKE 子句中避免使用通配符作为前导字符，例如 LIKE '%a%'。因为第一个字符是不确定的，所以，查询处理器无法使用可用的索引。应改用 LIKE 'a%'。
8、使用 Transact-SQL TRY...CATCH 功能进行过程内的错误处理
9、创建/修改 Table 时给列一个 Default 值

10、使用 IF EXISTS (SELECT TOP 1 FROM table_name) 来代替 IF EXISTS (SELECT * FROM table_name)。

11、使用 UNION ALL 运算符来代替 UNION 或 OR 运算符

12、

创建存储过程时在其定义中指定 WITH RECOMPILE 选项，表明 SQL Server 将不对该存储过程计划进行高速缓存；该存储过程将在每次执行时都重新编译。

当存储过程的参数值在各次执行间都有较大差异(个人觉得应该是动态创建存储过程时，使得很可能有时候参数不同)，导致每次均需创建不同的执行计划时，可使用 WITH RECOMPILE 选项。此选项并不常用，因为每次执行存储过程时都必须对其进行重新编译，这样会使存储过程的执行变慢，

二、知识点：

1、SQL Server 启动时可以自动执行一个或多个过程。这些过程必须由系统管理员在 **master** 数据库中创建，并以 **sysadmin** 固定服务器角色作为后台进程执行。这些过程不能有任何输入或输出参数。

2、过程可以嵌套，并且最多可以嵌套 32 级。

3、过程可以引用尚不存在的表。在创建时，只进行语法检查。直到第一次执行该过程时才对其进行编译。只有在编译过程中才解析过程中引用的所有对象。

二、返回多个值的存储过程

```
/*
作者：梁勇
时间：-1-5
描述：用一个字段去查询记录，如果不空就查询出结果，否则就显示'获取的数据为空'
*/
use City;
go
set nocount on
if OBJECT_ID(N'FirstP',N'P') is not null drop proc FirstP
go
Create procedure FirstP
  @CityName nvarchar(20),                  --输入参数(注意有个逗号)
  @CityPopulation nvarchar(30) output,  --输出参数，要指定关键字output
  @ProvinceName nvarchar(30) output      --有多个output用逗号隔开
  as --必备标示符
  set nocount on---好习惯
  select
  @CityPopulation=CityPopulation,
  @ProvinceName=ProvinceName--好像在有进行赋值的时候，那么这个select只能
select这个字段，不能select其他字段
         from CityTable
         where CityName=@CityName
go--】提交这个创建的批处理】
```

```
--执行存储过程

declare @provincename nvarchar(30)
set @provincename=''
declare @citypopulation nvarchar(30)
set @citypopulation=''

exec FirstP @CityName='s',
     @ProvinceName=@provincename output,--这里的参数必须和定义中的参数名称一样
     @CityPopulation=@citypopulation output--如果有多个output用逗号隔开

if @provincename<>''
    begin
       select * from CityTable where ProvinceName=@provincename
    end
else
    begin
        print '获取的数据为空'
    end
go--【提交这个访问的批处理】

/*问题：

*/
```

三、使用 Like 模糊查找的存储过程

```
/*
描述：在模糊查询中不使用通配符作为前导字符，以避免模糊查找的字段若是主键
      那么将失去索引的功效
*/
use City;
go
set nocount on
if OBJECT_ID(N'FirstP',N'P') is not null drop proc FirstP
go
Create procedure FirstP
  @CityName nvarchar(20)='1%',
  @ProvinceName nvarchar(30)='1%'
```

```
as
   set nocount on
   select C.bh,C.product_number  from C
   where C.product_number like @CityName
   and C.bh like @ProvinceName
go
exec FirstP--使用默认的值
exec FirstP @CityName='f%',@ProvinceName='3%'
drop proc FirstP
/*
bh         product_number
---------- --------------------
11         111112140503
11         111112140504
11         111112140404


bh         product_number
---------- --------------------
324        fdsfds
324        fsdfsdfds
324        fsdfdsfdsf
324        fsdffdssdfsfr3
*/
```

四、返回多记录的结果值

```
/*
描述：多记录的结果值
*/
use City;
go
set nocount on
if OBJECT_ID(N'FirstP',N'P') is not null drop proc FirstP
go
Create procedure FirstP
as
   set nocount on
   select * from C
go
exec FirstP
drop proc FirstP
```

数据库项目文档

```
/*
ID          product_number      bh          count1
----------- ------------------- ----------- -----------
1           111112140503        11          216
2           111112140504        11          171
3           111112140404        11          116
14          fdsfds              324         23
15          fsdfsdfds           324         23
16          fsdfdsfdsf          324         23
17          fsdffdssdfsfr3      324         23
18          hghgh               324         23
19          hhghghgf            324         23
*/
```

五、SQL查看存储过程原定义

```
Create procedure FirstP
as
   set nocount on
   select * from C
go
EXEC sp_helptext 'dbo.FirstP'
/*
Create procedure FirstP
as
   set nocount on
   select * from C
*/
```

或者

```
SELECT *
FROM sys.sql_modules
WHERE object_id = OBJECT_ID('dbo.FirstP');
```

六、加密存储过程

```
Create procedure FirstP
WITH ENCRYPTION
as
   set nocount on
   select * from C
go
EXEC sp_helptext 'dbo.FirstP'
/*
```

```
对象'dbo.FirstP' 的文本已加密。
*/
```

七、比较常见的添加、删除、修改的存储过程

1、添加

```
Create procedure AddTeacherCourse
@Name varchar(50),
@TeacherID int,
@CourseID int,
@CourseTime varchar(50)
as
  select ID from TeacherCourse where Name=@Name
  if(@@rowcount>=1)
  return -1--该信息已经存在
  Insert TeacherCourse
  values
  (
   @Name,
   @TeacherID,
   @CourseID,
   @CourseTime
  )
  if(@@error<>0)
    return 0--添加错误
  else
    return 1--添加成功
```

2、删除

```
create procedure Dele
@name varchar(50),
@id int
as
  if(@name='上课教师基本信息')
   begin
    select * from Teacher where ID=@id
    if(@@rowcount<1)
    return -1 --不存在这条信息
    delete from Teacher where ID=@id
    IF @@ERROR <> 0
       RETURN 0  --删除时发生错误
     ELSE
```

```
        RETURN 1 --删除成功
    End

3、更新
create procedure UpTeacherCourse
@ID int,
@Name varchar(50),
@TeacherID int,
@CourseID int,
@CourseTime varchar(50)
as
 select * from Teacher where ID=@ID
 if(@@rowcount<1)
 return -1--不存在这条信息
 update UpTeacherCourse
 set
   Name=@Name,
   TeacherID=@TeacherID,
   CourseID=@CourseID,
   CourseTime=@CourseTime
  where(ID=@ID)
  if(@@error<>0)
  return 0--更新发生错误
  else
  return 1--更新成功


use City;
go
if exists(select * from sysobjects where name='FirstP' and type='P')
    begin
        drop procedure FirstP
       print '存在存储过程FirstP,现已删除！'
    end
go--【先提交这个删除的批处理】
Create procedure FirstP
  @CityName='默认值' nvarchar(20),       --输入参数(注意有个逗号)
  @CityPopulation nvarchar(30) output, --输出参数，要指定关键字output
  @ProvinceName nvarchar(30) output--有多个output用逗号隔开
  as --必备标示符
  select
```

```
  @CityPopulation=CityPopulation,
  @ProvinceName=ProvinceName--好像在有进行赋值的时候，那么这个select只能
select这个字段，不能select其他字段
         from CityTable
         where CityName=@CityName
go--】提交这个创建的批处理】
declare @provincename nvarchar(30)
set @provincename=''
declare @citypopulation nvarchar(30)
set @citypopulation=''

exec FirstP @CityName='南平',
     @ProvinceName=@provincename output,--这里的参数必须和定义中的参数名称一
样
     @CityPopulation=@citypopulation output--如果有多个output用逗号隔开

if @provincename<>''
    begin
       select * from CityTable where ProvinceName=@provincename
    end
else
    begin
        print '获取的数据为空'
    end
go--【提交这个访问的批处理】

--select * from sysobjects where name='FirstP'
--select * from CityTable

/*
问题：
不能用if删除存储过程else创建--会报错
*/
```

函数的应用

一、概念

1、存储过程可以很好地用于处理复杂的 SQL 逻辑、保证和控制对数据的访问，以及将行集返回到调用例程，无论此例程是基于 Visual Basic 的程序，还是另一个 Transact-SQL (T-SQL) 批处理文件。

2、因而视图常用于表示常用的 SELECT 语句，该语句可以联接多个表、使用 WHERE 子句，以及公开特定的列。

3、有两种主要的 UDF 类型：返回标量值的 UDF 和返回表值的 UDF(返回内联表值的 UDF 和返回多语句表值的 UDF)。

4、UDF 无法返回下列数据类型中任何一个的值：text、ntext、image、cursor、或 timestamp。

5、UDF 是使用由对象所有者和对象名这两个部分组成的名称调用的：

```
    SELECT     ProductID,
     ReorderLevel,
     UnitsInStock,
     UnitsOnOrder,
     dbo.fnNeedToReorder(ReorderLevel, UnitsInStock, UnitsOnOrder)
AS sNeedToReorder
FROM     Products
```

7、个人感觉，udf 就是不管是在存储过程也好，还是平时的 sql 批处理语句中也好，如果出现一个小逻辑功能用到了两次以上那么就可以将其写成一个 udf，比如：格式化函数、或者写一个函数自增主键不为整型等。Udf 个人感觉是给数据库这块的逻辑代码调用（如供存储过程和触发器调用），而像存储过程和触发器是供应该程序调用。想.net 的函数

二、实例

1、返回值为标量的实例

```
use City;
go
set nocount on
if OBJECT_ID(N'FirstFN',N'FN') is not null drop function dbo.FirstFN
go
create function dbo.FirstFN(@firstv int,@secondV int)
returns int--★★★returns
begin      --★★★begin
declare @maxV int
if @firstV>=@secondV
begin
   set @maxV=@firstV
end
else
begin
   set @maxV=@secondV
end
return @maxV
end      --★★★end
```

```
go
select dbo.FirstFN(433,65)
```

这本来可以通过 CASE 语句在 SELECT 子句内完成，但如果改为使用 UDF，代码就会简洁得多。而且更容易传播到其他可能需要相同逻辑的地方。假定一个应用程序中有几个部分需要确定是否要重新订购产品，那么上面 中的 UDF 确实变得有价值，因为它使得当逻辑改变时应用程序更容易维护，当然 udf 中也还是可以用 case when 语句的----------封装

2、内联表值 UDF

[1]、内联表值 UDF 和视图返回的表都是由一个查询来定义的，从这个意义来讲，它们很相似。然而，UDF 的查询可以有输入参数，而视图没有。所以，你可以把内联 UDF 看作是参数化的视图---1、和视图很相似

[2]、不同于标量 UDF 和多语句表值 UDF，2、不能在内联 UDF 主体内使用 BEGIN/END 块。用 as　go。3、只能指定一个 RETURN 子句和 4、一个查询。5、在函数头，只需声明它返回一个表即可。

[3]:

```
use City;
go
if OBJECT_ID(N'MyF')is not null drop function dbo.MyF
go
create function dbo.MyF(@v int)     -这是参数
returns table                       -注意这是returns
as                                  -只能是as开头
  return                            -只能包含一个return、即用括号将结果包围起来
  (
  select * from D where ID=@v   -只能是一条sql语句、但可以返回多条记录
  )
go                                  -只能是go结束

select * from dbo.MyF(6)          -注意加上Schema名称
drop function dbo.MyF
```

[4]、可以对该内联函数返回的表当做一个'伪基表'来进行选择，修改，添加和删除，不管是哪种操作，其实是对内联函数中引用的基表进行操作。

[5]、若参数为空，则返回值为空：returns null on null input

3、多语句表值 UDF。

[1]、在总体上有点类似于内联的表值 UDF。多语句表值 UDF 是一种返回表变量的函数

[2]、不用单条语句(内联)，用多条语句(多语句)

[3]、只能对多语句表 UDF 进行 select，不能 insert、update、delete

```
use City;
go
set nocount on
```

```
if OBJECT_ID(N'FirstFN',N'FN') is not null drop function dbo.FirstFN
go
create function dbo.FirstFN(@V int)
returns @myReturnTable table --多一个表名
(                          --多了定义
ID int,
pro nvarchar(100),
bh int
)
begin
insert into @myReturnTable   --给上面的表插入数据
select ID,product_number,bh from C where count1=@V
return                      --直接写一个rerurn即可
end
go
select * from dbo.FirstFN(23)--返回表集合
--select dbo.FirstFN(23)     --返回标量值
   drop function dbo.FirstFN
```

注意点：

1、创建和使用(调用和删除)UDF 和 Proc 的时候都要加上 Schema

2、UDF 中第二句是 returns

3、除了内联函数是 As go 外，返回标量值的 UDF 和 TVF 都是 Begin end

4、内联 UDF,return （一条 SQL 语句)，但结果集可以多条，且 UDF 结果可以用于 DML 操作。

5、TVF:首先要声明 returns @MyReturnTable table(id int…)

Begin

insert into @MyReturnTable select ……

Return 一条语句结束即可

End

不能用于只能 Select,不能 DML 其他操作

触发器的应用

触发器是一种特殊类型的存储过程，它不同于之前的我们介绍的存储过程。触发器主要是通过事件进行触发被自动调用执行的。而存储过程可以通过存储过程的名称被调用。

Ø 什么是触发器

数据库项目文档

触发器对表进行插入、更新、删除的时候会自动执行的特殊存储过程。触发器一般用在 check 约束更加复杂的约束上面。触发器和普通的存储过程的区别是：触发器是当对某一个表进行操作。诸如：update、insert、delete 这些操作的时候，系统会自动调用执行该表上对应的触发器。SQL Server 2005 中触发器可以分为两类：DML 触发器和 DDL 触发器，其中 DDL 触发器它们会影响多种数据定义语言语句而激发，这些语句有 create、alter、drop 语句。

DML 触发器分为：

1、 after 触发器（之后触发）

a、 insert 触发器

b、 update 触发器

c、 delete 触发器

2、 instead of 触发器 （之前触发）

其中 after 触发器要求只有执行某一操作 insert、update、delete 之后触发器才被触发，且只能定义在表上。而 instead of 触发器表示并不执行其定义的操作（insert、update、delete）而仅是执行触发器本身。既可以在表上定义 instead of 触发器，也可以在视图上定义。

触发器有两个特殊的表：插入表（instered 表）和删除表（deleted 表）。这两张是逻辑表也是虚表。有系统在内存中创建者两张表，不会存储在数据库中。而且两张表的都是只读的，只能读取数据而不能修改数据。这两张表的结果总是与被改触发器应用的表的结构相同。当触发器完成工作后，这两张表就会被删除。Inserted 表的数据是插入或是修改后的数据，而 deleted 表的数据是更新前的或是删除的数据。

对表的操作	Inserted 逻辑表	Deleted 逻辑表
增加记录（insert）	存放增加的记录	无
删除记录（delete）	无	存放被删除的记录
修改记录（update）	存放更新后的记录	存放更新前的记录

Update 数据的时候就是先删除表记录，然后增加一条记录。这样在 inserted 和 deleted 表就都有 update 后的数据记录了。注意的是：触发器本身就是一个事务，所以在触发器里面可以对修改数据进行一些特殊的检查。如果不满足可以利用事务回滚，撤销操作。

Ø 创建触发器

语法

```
create trigger tgr_name
on table_name
with encrypion –加密触发器
    for update...
as
    Transact-SQL
```

创建 insert 类型触发器

```
--创建 insert 插入类型触发器
if (object_id('tgr_classes_insert', 'tr') is not null)
    drop trigger tgr_classes_insert
go
create trigger tgr_classes_insert
on classes
    for insert --插入触发
as
    --定义变量
    declare @id int, @name varchar(20), @temp int;
    --在 inserted 表中查询已经插入记录信息
    select @id = id, @name = name from inserted;
    set @name = @name + convert(varchar, @id);
    set @temp = @id / 2;
    insert into student values(@name, 18 + @id, @temp, @id);
    print '添加学生成功！';
go
--插入数据
insert into classes values('5 班', getDate());
--查询数据
select * from classes;
select * from student order by id;
```

insert 触发器，会在 inserted 表中添加一条刚插入的记录。

创建 delete 类型触发器

```
--delete 删除类型触发器
if (object_id('tgr_classes_delete', 'TR') is not null)
    drop trigger tgr_classes_delete
go
create trigger tgr_classes_delete
on classes
    for delete --删除触发
as
    print '备份数据中……';
    if (object_id('classesBackup', 'U') is not null)
        --存在 classesBackup，直接插入数据
        insert into classesBackup select name, createDate from deleted;
    else
        --不存在 classesBackup 创建再插入
        select * into classesBackup from deleted;
    print '备份数据成功！';
go
--
--不显示影响行数
--set nocount on;
delete classes where name = '5 班';
--查询数据
select * from classes;
select * from classesBackup;
```

delete 触发器会在删除数据的时候，将刚才删除的数据保存在 deleted 表中。

创建 update 类型触发器

```
--update 更新类型触发器
if (object_id('tgr_classes_update', 'TR') is not null)
    drop trigger tgr_classes_update
go
create trigger tgr_classes_update
```

```
on classes
    for update
as
    declare @oldName varchar(20), @newName varchar(20);
    --更新前的数据
    select @oldName = name from deleted;
    if (exists (select * from student where name like '%'+ @oldName + '%'))
        begin
            --更新后的数据
            select @newName = name from inserted;
            update student set name = replace(name, @oldName, @newName)
where name like '%'+ @oldName + '%';
            print '级联修改数据成功！';
        end
    else
        print '无需修改 student 表！';
go
--查询数据
select * from student order by id;
select * from classes;
update classes set name = '五班' where name = '5 班';
```

update 触发器会在更新数据后，将更新前的数据保存在 deleted 表中，更新后的数据保存在 inserted 表中。

update 更新列级触发器

```
if (object_id('tgr_classes_update_column', 'TR') is not null)
    drop trigger tgr_classes_update_column
go
create trigger tgr_classes_update_column
on classes
    for update
as
    --列级触发器：是否更新了班级创建时间
    if (update(createDate))
    begin
        raisError('系统提示：班级创建时间不能修改！', 16, 11);
        rollback tran;
```

```
    end
go
--测试
select * from student order by id;
select * from classes;
update classes set createDate = getDate() where id = 3;
update classes set name = '四班' where id = 7;
```

更新列级触发器可以用 update 是否判断更新列记录；

instead of 类型触发器

instead of 触发器表示并不执行其定义的操作（insert、update、delete）而仅是执行触发器本身的内容。

创建语法

```
create trigger tgr_name
on table_name
with encryption
    instead of update...
as
    T-SQL
```

创建 instead of 触发器

```
if (object_id('tgr_classes_inteadOf', 'TR') is not null)
    drop trigger tgr_classes_inteadOf
go
create trigger tgr_classes_inteadOf
on classes
    instead of delete/*, update, insert*/
as
    declare @id int, @name varchar(20);
    --查询被删除的信息，病赋值
    select @id = id, @name = name from deleted;
    print 'id: ' + convert(varchar, @id) + ', name: ' + @name;
    --先删除 student 的信息
```

```
    delete student where cid = @id;
    --再删除 classes 的信息
    delete classes where id = @id;
    print '删除[ id: ' + convert(varchar, @id) + ', name: ' + @name + ' ]
的信息成功! ';
go
--test
select * from student order by id;
select * from classes;
delete classes where id = 7;
```

显示自定义消息 raiserror

```
if (object_id('tgr_message', 'TR') is not null)
    drop trigger tgr_message
go
create trigger tgr_message
on student
    after insert, update
as raisError('tgr_message 触发器被触发', 16, 10);
go
--test
insert into student values('lily', 22, 1, 7);
update student set sex = 0 where name = 'lucy';
select * from student order by id;
```

修改触发器

```
alter trigger tgr_message
on student
after delete
as raisError('tgr_message 触发器被触发', 16, 10);
go
--test
delete from student where name = 'lucy';
```

启用、禁用触发器

```
--禁用触发器
disable trigger tgr_message on student;
--启用触发器
enable trigger tgr_message on student;
```

```
   # 查询创建的触发器信息
```

```
--查询已存在的触发器
select * from sys.triggers;
select * from sys.objects where type = 'TR';

--查看触发器触发事件
select te.* from sys.trigger_events te join sys.triggers t
on t.object_id = te.object_id
where t.parent_class = 0 and t.name = 'tgr_valid_data';

--查看创建触发器语句
exec sp_helptext 'tgr_message';
```

```
   # 示例，验证插入数据
```

```
if ((object_id('tgr_valid_data', 'TR') is not null))
    drop trigger tgr_valid_data
go
create trigger tgr_valid_data
on student
after insert
as
    declare @age int,
            @name varchar(20);
    select @name = s.name, @age = s.age from inserted s;
    if (@age < 18)
    begin
        raisError('插入新数据的 age 有问题', 16, 1);
        rollback tran;
    end
go
--test
insert into student values('forest', 2, 0, 7);
```

```
insert into student values('forest', 22, 0, 7);
select * from student order by id;
```

示例，操作日志

```
if (object_id('log', 'U') is not null)
    drop table log
go
create table log(
    id int identity(1, 1) primary key,
    action varchar(20),
    createDate datetime default getDate()
)
go
if (exists (select * from sys.objects where name = 'tgr_student_log'))
    drop trigger tgr_student_log
go
create trigger tgr_student_log
on student
after insert, update, delete
as
    if ((exists (select 1 from inserted)) and (exists (select 1 from
deleted)))
    begin
        insert into log(action) values('updated');
    end
    else if (exists (select 1 from inserted) and not exists (select 1 from
deleted))
    begin
        insert into log(action) values('inserted');
    end
    else if (not exists (select 1 from inserted) and exists (select 1 from
deleted))
    begin
        insert into log(action) values('deleted');
    end
go
--test
insert into student values('king', 22, 1, 7);
update student set sex = 0 where name = 'king';
```

```
delete student where name = 'king';
select * from log;
select * from student order by id;
```

一、truncate table 语句没有记录，不触发 delete 触发器
二、触发器作用在表或视图上
三、Create trggter name
On table/view
With encryption—加密 sysconnents 表中包含 create trigger 语句文本的条目
For|after|instead of—for 是默认的设置表示 after，after 表示在触发 sql 语句中指定的所有操作都已经成功执行后才激发【不能在视图上定义 after 触发器】，instead of 表示只执行触发器，而不执行触发器的 sql 语句【在 insert|delete|update 中只能定义一个这样的 instead of】
Insert|delete|update
As
—触发器要执行的触发操作,如 print '操作成功'
Go
Sql_statement—触发的条件和操作，如插入一条语句或者其他

游标的应用

SQl Server的游标在你需要对记录集进行单条处理时很有用处。
select返回一个记录集，但是你想根据每条记录的不同情况进行单条处理，这时游标的用处就显出来了。

一个游标(cursor)可以被看作指向结果集(a set of rows)中一行的指针(pointer)。游标每个时间点只能指向一行，但是可以根据需要指向结果集中其他的行。
例如：SELECT * FROM employees WHERE sex='M'会返回所有性别为男的雇员，在初始的时候，游标被放置在结果集中第一行的前面。使游标指向第一行，要执行FETCH。当游标指向结果集中一行的时候，可以对这行数据进行加工处理，要想得到下一行数据，要继续执行FETCH。FETCH操作可以重复执行，直到完成结果集中的所有行

在存储过程中使用游标，有如下几个步骤：
声明游标、打开游标、根据需要一次一行，讲游标指向的数据取到本地变量(local variables)中、结束时关闭游标

创建游标分五个步骤：

数据库项目文档

```
-- =============================================
-- Author:      Zehui Shu
-- Create date: 2010-12-23
-- Description: 通过游标实现同步表数据
-- =============================================
CREATE PROCEDURE [dbo].[Proc_Syn_Data]
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
    DECLARE @ID INT;
    TRUNCATE TABLE Table2;



    -- 1.声明游标
    DECLARE CUR_MonthID CURSOR FOR
    SELECT ID FROM [Table1]

    -- 2.打开游标
    OPEN CUR_MonthID
    -- 3.从一个游标中查找信息，实现自己的数据处理。
    FETCH CUR_MonthID INTO @ID
    WHILE @@FETCH_STATUS=0
    BEGIN
        INSERT INTO [Table2]
        SELECT * FROM [Table1] tm WHERE tm.ID=@ID
        FETCH NEXT FROM CUR_MonthID INTO @ID
    END;

    -- 4.关闭游标
    CLOSE CUR_MonthID;

    -- 5.释放游标
    DEALLOCATE CUR_MonthID;
END
游标 还是看下上篇的那段代码：
create proc pro_abc

as
```

```
    declare @p1 nvarchar(50),@p2 int;

    declare my_cursor cursor scroll dynamic

    for SELECT F_CR_JSGM,f_nb_xmbm FROM B_JIHUA_XIANGMU WHERE NOT
F_CR_JSGM IS NULL

       open my_cursor

       fetch next from my_cursor into @p1,@p2

       while(@@fetch_status=0)

         begin

           update b_jihua_jsgm set f_cr_jsgm2=@p1 where f_nb_xmbm=@p2

           fetch next from my_cursor into @p1,@p2

         end

       close my_cursor

       deallocate my_cursor
```

分解：这段是存储过程

```
    create proc pro_abc--

    as

    declare @p1 nvarchar(50),@p2 int;

         begin

          select 'aaaa'

         end
```

加上游标就是这样了

```
    create proc pro_abc
```

```
    as

    declare @p1 nvarchar(50),@p2 int;--声明变量和类型

    --声明游标的名称my_cursor；这个游标所用到的SQL语句
    view sourceprint?
    ----scroll表示可随意移动游标指        针（否则只能向前），dynamic表示可以读写游标（否则游标只读）*/
    declare my_cursor cursor scroll dynamic
    for SELECT F_CR_JSGM,f_nb_xmbm FROM B_JIHUA_XIANGMU WHERE NOT F_CR_JSGM IS NULL

    --打开游标

       open my_cursor

    --必须用FETCH语句来取得数据 可以传递参数

       fetch next from my_cursor into @p1,@p2

    --fetch_status=0一切正常

       while(@@fetch_status=0)

         begin

           update b_jihua_jsgm set f_cr_jsgm2=@p1 where f_nb_xmbm=@p2

           fetch next from my_cursor into @p1,@p2

         end

    --关闭游标

       close my_cursor

    --删除游标资源

       deallocate my_cursor
    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

数据库项目文档

声明游标

象使用其它类型的变量一样,使用一个游标之前,首先应当声明它。游标的声明包括两个部分:游标的名称;这个游标所用到的SQL语句

在游标的声明中有一点值得注意的是,如同其它变量的声明一样,声明游标的这一段代码行是不执行的,您不能将debug时的断点设在这一代码行上,也不能用IF...END IF语句来声明两个同名的游标,如下列的代码就是错误的。

```
IF Is_prov="北京"THEN
DECLARE CustomerCursor CURSOR FOR
SELECT acct_no,name,balance
FROM customer
WHERE province="北京";
ELSE
DECLARE CustomerCursor CURSOR FOR
SELECT acct_no,name,balance
FROM customer
WHERE province 〈〉 "北京";
END IF
```

由于打开游标是对数据库进行一些SQL SELECT的操作,它将耗费一段时间,主要取决于您使用的系统性能和这条语句的复杂程度。如果执行的时间较长,可以考虑将屏幕上显示的鼠标改为hourglass(这句不会)

提取数据

当用OPEN语句打开了游标并在数据库中执行了查询后,您不能立即利用在查询结果集中的数据。您必须用FETCH语句来取得数据。一条FETCH语句一次可以将一条记录放入程序员指定的变量中。事实上,FETCH语句是游标使用的核心。在DataWindow和DataStore中,执行了Retrieve()函数以后,查询的所有结果全部可以得到;而使用游标,我们只能逐条记录地得到查询结果。

已经声明并打开一个游标后,我们就可以将数据放入任意的变量中。在FETCH语句中您可以指定游标的名称和目标变量的名称

从语法上讲,上面所述的就是一条合法的取数据的语句,但是一般我们使用游标却还应当包括其它的部分。正如我们前面所谈到的,游标只能一次从后台数据库中取一条记录,而在多数情况下,我们所想要作的是在数据库中从第一条记录开始提取,一直到结束。所以我们一般要将游标提取数据的语句放在一个循环体内,直至将结果集中的全部数据提取后,跳出循环圈。通过检

测SQLCA.SQL-CODE的值,可以得知最后一条FETCH语句是否成功。一般,当SQLCODE值为0时表明一切正常,100表示已经取到了结果集的末尾,而其它值均表明操作出了问题,

更多的就是这里了http://www.knowsky.com/344071.html

今天就学到这里 再见

----------男人如车，女人如房。

SQL 游标分类

静态游标

静态游标（Static Cursor）是利用暂存资料表作为储存结果集空间的一种游标，它可以让应用程式可以快速的存取结果集，但在静态游标开启期间，任何对资料表所做的变更都不会反映在结果集中；同时，在静态游标中所作的修改，无法反映到资料库中，此种游标是消耗资源度第三的游标。

Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in tempdb; therefore, modifications made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow modifications.

动态游标

动态游标（Dynamic Cursor）是可以反映资料库中修改的一种游标，不过它并不会让结果集的位置固定（随机变动），因此无法确实的以游标位置来判断资料，并且它因为要随时反映资料库的变化，因此伺服器需要消耗较多的资源，此种游标是消耗资源第二高的游标。

Defines a cursor that reflects all data changes made to the rows in its result set as you scroll around the cursor. The data values, order, and membership of the rows can change on each fetch. The ABSOLUTE fetch option is not supported with dynamic cursors.

索引键集型游标

索引键集游标（Keyset Cursor）是动态游标的强化版本，借由维护一个资料集位置对应表（以 SQL Server 为例，会建立在 tempdb 的 keyset 资料表中），以维护在结果集中的顺序不受更新而变化，但这相对的也付出了伺服器效能和资源消耗的代价，因此索引键集游标是最消耗伺服器资源的一种游标，在实务上应避免使用。

仅前移型游标

仅前移型游标（Forward-Only Cursor）是一旦将游标往前移时，其走过的游标之前的结果集就会被舍弃，因此应用程式不能再往后移动游标，但也因此让伺服器只需要记住游标在结果集中目前的位置即可，这让它消耗的资源只有游标而已，是最省资源的一种游标，在实务中被广泛使用，像 ADO.NET 的 DataReader 就只限定只能使用 Forward-Only Cursor。

Specifies that the cursor can only be scrolled from the first to the last row. FETCH NEXT is the only supported fetch option. If FORWARD_ONLY is specified without the STATIC, KEYSET, or DYNAMIC keywords, the cursor operates as a DYNAMIC cursor. When neither FORWARD_ONLY nor SCROLL is specified, FORWARD_ONLY is the default, unless the keywords STATIC, KEYSET, or DYNAMIC are specified. STATIC, KEYSET, and DYNAMIC cursors default to SCROLL. Unlike database APIs such as ODBC and ADO, FORWARD_ONLY is supported with STATIC, KEYSET, and DYNAMIC Transact-SQL cursors.

5.SqlServer 查询计划，性能优化，分布式查询

SQL 优化

sql 语句优化背后有一套完整的理论和逻辑，这里只是给你一个思路：

1：确认是否因为做了物理 I/O 而导致性能不佳

2：确认是否因为编译时间过长而导致性能不佳

3：确认 sql server 是否正确估计了每一步的 cast，选择查询计划，

4：检查表结构和语句逻辑，确认是否有优化空间，提交执行速度。

每一条 sql 执行都要经过编译,执行和返回结果集的阶段。在编译阶段需要查看缓存中是否缓存了查询计划，如果没有就要生成新的查询计划，所以在排除了 IO，内存，网络等情况后 sql 调优就是看你能不能读懂查询计划了

给你提供一个语句查看一些统计信息：

SQL code

--查看查询计划

```
    set statistics profile on
    go
    select * from dbo.authors
    go
    set statistics profile off
    go


    --查看 IO 读取情况
    SET STATISTICS IO
    go
    select * from dbo.authors
    go
    SET STATISTICS off
    go

    --查看编译，执行，和返回结果集的时间
    SET STATISTICS TIME ON
    go
    select * from dbo.authors
    go
SET STATISTICS TIME OFF

    ---使用索引优化数据库查询效率
    1.不宜创建索引的情形
    (1)经常插入，修改和删除的表
    (2)数据量比较小的表，因为查询优化器在搜索索引时所花费的时间可能会大于遍历全表的数据所需要的时间

    2.适合创建索引的情形
    (1)为 where 子句中出现的列创建索引
    (2)创建组合索引
    (3)为 group by 子句中出现的列创建索引

    3.聚集索引的设计原则
    (1)该列的数值是唯一的或者很少有重复的记录
    (2)经常使用 between ...and..按顺序查询的列
    (3)定义 identity 的唯一列.
    (4)经常用于对数据进行排序的列.

    ---无法使用索引的 select 语句
```

1.对索引列使用了函数,如：

```
select * from tb where max(id)=100
```

2.对索引列使用了'%xx', 如：

```
select * from tb where id like '%1'
```

需要注意的不是所有使用 like 关键字的 select 语句都无法使用索引,比如

```
select * from tb where id like '1%'
```

就可以使用索引

3.在 where 子句中对列进行类型转换(其实也是使用到了函数)

4.在组合索引的第 1 列不是使用最多的列,如在下面 3 个查询语句中建立组合索引,按顺序包含 col2,col1,id 列：

```
select * from tb where id='1' and col1='aa'
select id,sum(col1) from tb group by id
select * from tb where id='2' and col2='bb'
```

则第一句和第二句无法使用到索引 所以需要注意组合索引的顺序

5.在 where 子句中使用 in 关键字的某些句子

当在 in 关键字后面使用嵌套的 select 语句，将无法使用在该列上定义的索引

如：

```
select
 *
from
 ta
where
 id
in
 (select id from tb where ....)

--这样可以用到索引
select * from tb where id in('1','2')
```

SQL 锁机制

一．为什么要引入锁

多个用户同时对数据库的并发操作时会带来以下数据不一致的问题：

丢失更新

数据库项目文档

A,B两个用户读同一数据并进行修改,其中一个用户的修改结果破坏了另一个修改的结果,比如订票系统

脏读
A用户修改了数据,随后B用户又读出该数据,但A用户因为某些原因取消了对数据的修改,数据恢复原值,此时B得到的数据就与数据库内的数据产生了不一致

不可重复读
A用户读取数据,随后B用户读出该数据并修改,此时A用户再读取数据时发现前后两次的值不一致

并发控制的主要方法是封锁,锁就是在一段时间内禁止用户做某些操作以避免产生数据不一致

二 锁的分类

锁的类别有两种分法：

1．从数据库系统的角度来看：分为独占锁（即排它锁），共享锁和更新锁

下例将锁超时期限设置为 1,800 毫秒。
```
SET LOCK_TIMEOUT 1800
```

1、几个例子帮助大家加深印象
设table1(A,B,C)
A B C
a1 b1 c1
a2 b2 c2
a3 b3 c3

1）排它锁
新建两个连接
在第一个连接中执行以下语句
```
begin tran
update table1
set A='aa'
where B='b2'
waitfor delay '00:00:30' --等待秒
commit tran
```
在第二个连接中执行以下语句
```
begin tran
select * from table1
```

```
where B='b2'
commit tran
```

若同时执行上述两个语句，则select查询必须等待update执行完毕才能执行即要等待秒

2）共享锁

在第一个连接中执行以下语句

```
begin tran
select * from table1 holdlock -holdlock人为加锁
where B='b2'
waitfor delay '00:00:30' --等待秒
commit tran
```

在第二个连接中执行以下语句

```
begin tran
select A,C from table1
where B='b2'
update table1
set A='aa'
where B='b2'
commit tran
```

若同时执行上述两个语句，则第二个连接中的select查询可以执行

而update必须等待第一个事务释放共享锁转为排它锁后才能执行 即要等待秒

3）死锁

增设table2(D,E)

```
D E
d1 e1
d2 e2
```

在第一个连接中执行以下语句

```
begin tran
update table1
set A='aa'
where B='b2'
waitfor delay '00:00:30'
update table2
set D='d5'
where E='e1'
commit tran
```

在第二个连接中执行以下语句

```
begin tran
update table2
set D='d5'
where E='e1'
waitfor delay '00:00:10'
update table1
set A='aa'
where B='b2'
commit tran
```

同时执行，系统会检测出死锁，并中止进程

补充一点：
Sql Server2000支持的表级锁定提示

HOLDLOCK 持有共享锁，直到整个事务完成，应该在被锁对象不需要时立即释放，等于SERIALIZABLE事务隔离级别

NOLOCK 语句执行时不发出共享锁，允许脏读 ，等于 READ UNCOMMITTED事务隔离级别

PAGLOCK 在使用一个表锁的地方用多个页锁

READPAST 让sql server跳过任何锁定行，执行事务，适用于READ UNCOMMITTED事务隔离级别只跳过RID锁，不跳过页，区域和表锁

ROWLOCK 强制使用行锁

TABLOCKX 强制使用独占表级锁，这个锁在事务期间阻止任何其他事务使用这个表

UPLOCK 强制在读表时使用更新而不用共享锁

应用程序锁：
应用程序锁就是客户端代码生成的锁，而不是sql server本身生成的锁

处理应用程序锁的两个过程

sp_getapplock 锁定应用程序资源

sp_releaseapplock 为应用程序资源解锁

数据库项目文档

注意：锁定数据库的一个表的区别

SELECT * FROM table WITH (HOLDLOCK) 其他事务可以读取表，但不能更新删除

SELECT * FROM table WITH (TABLOCKX) 其他事务不能读取表,更新和删除

SQL code

1 如何锁一个表的某一行

A 连接中执行

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

begin tran

select * from tablename with (rowlock) where id=3

waitfor delay '00:00:05'

commit tran
```

B连接中如果执行

```
update tablename set colname='10' where id=3 --则要等待秒

update tablename set colname='10' where id<>3 --可立即执行
```

2 锁定数据库的一个表

```
SELECT * FROM table WITH (HOLDLOCK)
```

注意：锁定数据库的一个表的区别

```
SELECT * FROM table WITH (HOLDLOCK)
```

其他事务可以读取表，但不能更新删除

```
SELECT * FROM table WITH (TABLOCKX)
```

其他事务不能读取表，更新和删除

共享锁

共享锁（S 锁）允许并发事务在封闭式并发控制下读取 (SELECT) 资源。

更新锁

更新锁（U 锁）可以防止常见的死锁。 在可重复读或可序列化事务中，此事务读取数据 [获取资源（页或行）的共享锁（S 锁）]，然后修改数据 [此操作要求锁转换为排他锁（X 锁）]。 如果两个事务获得了资源上的共享模式锁，然后试图同时更新数据，则一个事务尝试将锁转换为排他锁（X 锁）。 共享模式到排他锁的转换必须等待一段时间，因为一个事务的排他锁与其他事务的共享模式锁不兼容；发生锁等待。 第二个事务试图获取排他锁（X 锁）以进行更新。 由于两个事务都要转换为排他锁（X 锁），并且每个事务都等待另一个事务释放共享模式锁，因此发生死锁。

若要避免这种潜在的死锁问题，请使用更新锁（U 锁）。 一次只有一个事务可以获得资源的更新锁（U 锁）。 如果事务修改资源，则更新锁（U 锁）转换为排他锁（X 锁）。

排他锁

排他锁（X 锁）可以防止并发事务对资源进行访问。 使用排他锁（X 锁）时，任何其他事务都无法修改数据；仅在使用 NOLOCK 提示或未提交读隔离级别时才会进行读取操作。

数据修改语句（如 INSERT、UPDATE 和 DELETE）合并了修改和读取操作。 语句在执行所需的修改操作之前首先执行读取操作以获取数据。 因此，数据修改语句通常请求共享锁和排他锁。 例如，UPDATE 语句可能根据与一个表的联接修改另一个表中的行。 在此情况下，除了请求更新行上的排他锁之外，UPDATE 语句还将请求在联接表中读取的行上的共享锁。

SQL 索引

1、聚集索引：索引树的叶级页包含实际的数据：记录的索引顺序与物理顺序相同
2、非聚集索引：叶级页指向表中的记录，记录的物理顺序与逻辑顺序没有必然的联系
3、我们可以这么理解聚簇索引：索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块

而它在检索的性能损失也不会太大

所以能不用聚簇当然是最好的了
但是如果使用\order by 的话
聚簇的优势也应该是很明显的

每个表只能有一个聚簇索引，因为一个表中的记录只能以一种物理顺序存放。通常你要对一个表按照标识字段建立聚簇索引

为什么索引快？
1、SQL Server 的最小空间分配单元是“页（Page）”。
2、假设找最后一条，由于索引快的大小比单条记录的大小要下很多，那么每页存储的索引快就多，那么也许只要访问几个页面就找到了目标数据，即减少了 IO 访问量。
3、假设找所有的数据，那么索引就未必快了。比如检索所有的数据，那么在不用索引(即用表扫描)的情况下，可能要访问 1000 个页面。然后用索引要先访问 10 个页面找到索引项，然后在根据这些索引项再访问 1000 个页面访问到数据，即用所有总共访问 1010 个页面。
4、SQL Server 内部有一套完整的数据检索优化技术，在上述 3 的情况下，SQL Server 的查询计划（Search Plan）会自动使用表扫描的方式检索数据而不会使用任何索引。因为 SQL Server 除了日常维护数据信息外，还维护着数据统计信息
5、个人理解，SQL 内部在查询有索引的字段时，根据某种算法权衡了下到底是用索引要访问的页面数和用表扫描要访问的页面数进行了比较，小的那个即为最后的选择

聚集索引和非聚集索引的索引节点和数据节点的算法：
1、聚集索引的叶级节点包含两个部分，一个索引节点(N 个)(10 个页面大小)，一个是数据节点(N 个)(1000 个页面大小)。
2、非聚集索引的叶级节点包含两部分，一个索引节点(N 个),一个是指向数据的指针节点(N 个)，即索引节点为 2N 个（20 个页面大小）。另外还包含 N 个数据节点（1000 个页面大小）
3、于是插入数据的时候判断算法有重复的时候，那么非聚集索引由于在 2N 个索引节点中就包含了主键的值，所有只要访问 20 个页面，而聚集索引的主键是包含在数据节点中，那么就要访问 1000+10=1010 个页面才能知道是否重复。当然另外的页拆分也是耗效率的地方，即插入数据的时候非聚集索引更快

聚集索引和非聚集索引检索数据的速度，是访问所有的数据
1、聚集索引最差的情况是 10 个索引节点页+1000 个数据节点页=1010 个页面(通常少于 1000，最优)
2、非聚集索引是 20 个索引节点页+1000 个数据节点页=1020 个页面
3、而表扫描的方式只要 1000 个数据节点页

对每个表你最多可以建立 249 个非聚簇索引

数据库项目文档

一、测试显示计划中 Table Scan、Index Scan、Index Seek、Clusted Index Scan、Clusted Index Seek 的应用范围

1、没有索引的表所有操作为 Table Scan

2、有非聚集索引的表：

1、Select 列很多，但是最后结果很少时 Index Seek，记录大时 Table Scan

2、select 只有非聚集索引列:

1、无条件或者条件只要非聚集索引 Index Scan

2、条件有其他的字段，数量小 Index Scan，数量大 Table Scan

总结：堆上的非聚集索引只有 1 种情况是用到 Index Scan:

1、 select 非聚集索引列，无 where

堆上的非聚集索引只有 1 种情况是用 Index Seek

2、 select 非聚集索引列，where 也只为非聚集列

3、 除了上面 1 和 2 的其他任何情况行数少的话有可能用到 Index Seek

堆上的非聚集索引必用到 Table Scan

1、 有 where 条件，且条件中没有非聚集索引

3、聚集索引的表

只有两种情况是 Clustered Index Scan，其他情况都为 Clustered Index Seek.

1、 无 where 语句

2、 有 where 条件、且 where 语句不含聚集索引列

二、在书上的笔记

1、非聚集索引的叶子节点中的那个指针是指 RID:特定行的区段、页和行偏移量

2、有索引时，在对字段进行更新时是先进行删除，再进行添加的

3、迁移聚集索引其实就是在迁移数据。

4、当索引是复合索引时，在 where 中必须要包含第一个索引列，其余的可以不包含也可以包含。

5、在对建立了非聚集索引的聚集表上进行查询时：

1、先查找非聚集索引，找到叶子节点，此时叶子节点是聚集索引的位置

2、查找聚集索引的叶子节点即数据

6、聚集索引赞成：

1、那些经常作为查找范围的对象，如 between、or、group by、order by 等

2、索引键是连续的(比如我们的 ID identity(1,1)),插入的时候有两种情况，第一种：不用页拆分，直接插入到页的后面；第二种：需要页拆分，那么直接添加新页，并且把新纪录添加到新页中，无需对旧数据进行迁移。这样页拆分的整体耗率也不是很高，但是要找是否重复的键这个过程(本文上面介绍的)比较耗效率。

不过这种情况要注意一点，就是并发生成聚集索引键的问题，比如类似 001、002、003 等这种数据格式作为聚集索引键时要判断好在数据库中不出现两个相同的值索引键的记录。

聚集索引反对：

1、索引键不是连续的、这样在进行插入的时候肯定要进行页拆分，并且涉及旧数据的迁移。大大耗效率。如果一个系统的 select>>insert 这种方式还是可以的

2、此时可以改为非聚集索引键，如果是 insert>>select 那么只能这样了。

其实还有一种极端的方法，就是先将插入的数据放在一个临时表中，这个临时表中没有任何索引，等到晚上或者什么时候再统一插入到正式数据库中。这样避开用户选择数据的高分期，当然这个系统必须不要求数据的实时性。即即插即可查性。

SQL 性能优化

一、执行计划

1、 SQL Server 把批处理中的语句编译到一个被称为“执行计划”(execution plan)的可执行单元

2、 编译和执行查询处理是两个截然不同的阶段，有可能一个编译需要几个小时，可是执行该查询只需要几毫秒。处理即时查询时缓存中通常不包含其执行计划，因此它被编译后会立即执行。而经常执行的存储过程的已编译计划可能会在过程缓存中保留很长时间。在需要释放存储空间时，SQL Server 会从过程缓存中首先移除不常用的计划

3、 3、当 SQL Server 准备处理一个批处理时，如果缓存中不存在该批处理的执行计划，则编译该批处理并生成执行计划：

   1、 **SQL Server 执行分析：**分析是检查语法并把 SQL 批处理转换为分析树(parse tree)的过程

   2、 **SQL Server 进行绑定。**绑定过程确定 SQL 语句所引用对象的特征，他检查请求语义是否有意义

   3、 **优化：**优化器一次只优化批处理中的一条语句。在编译器为该批处理生成执行计划并存储到过程缓存之后，将执行该计划的执行上下文的一个特殊副本。SQL Server 像缓存查询计划一样缓存执行上下文。SQL Server 并不优化批处理中的每条语句。它只优化那些访问表而且可能生成多个执行计划的语句。SQL Server 优化所有的 DML 语句(SELECT、INSERT、DELETE、UPDATE)。除了 DML ，一些 T-SQL 语句也会被优化，CREATE INDEX 便是其中之一。只有被优化过的语句才会生成查询计划。

4、 执行计划(execution)是一个可执行单元、而显示计划(showplan)只是表示由查询优化器生成的文本、图形或 XML 格式的查询计划的术语

二、显示计划：

1、表示由查询优化器生成的文本、图形或 XML 格式的查询计划的术语

2、在计划内容方面，SQL Server 可以生成只包含运算符的计划，包含估计成本的计划，以及包含运行时信息的计划。

三、我觉得要提高一个查询的效率，最重要是用户的需求，即用户需求中哪个字段被作为条件来查找的频率最高，即在那个字段上建立聚集索引，比如时间等

四、测试显示计划中 Table Scan、Index Scan、Index Seek、Clusted Index Scan、Clusted Index Seek 的应用范围

1、没有索引的表所有操作为 Table Scan

2、有非聚集索引的表：

1、Select 列很多，但是最后结果很少时 Index Seek，记录大时 Table Scan

2、select 只有非聚集索引列:

1、无条件或者条件只要非聚集索引 Index Scan

2、条件有其他的字段，数量小 Index Scan，数量大 Table Scan

总结：堆上的非聚集索引只有 1 种情况是用到 Index Scan:

4、 select 非聚集索引列，无 where

堆上的非聚集索引只有 1 种情况是用 Index Seek

5、 select 非聚集索引列，where 也只为非聚集列

6、 除了上面 1 和 2 的其他任何情况行数少的话有可能用到 Index Seek

堆上的非聚集索引必用到 Table Scan

2、 有 where 条件，且条件中没有非聚集索引

3、聚集索引的表

只有两种情况是 Clustered Index Scan，其他情况都为 Clustered Index Seek.

3、 无 where 语句

4、 有 where 条件、且 where 语句不含聚集索引列

6.我要经常看的.

字符串相加类型全部相同

```
declare @p_code int , @sql nvarchar(max);
set @p_code=0;
set @sql=' select cast('+ltrim(@p_code)+'*100+5 as numeric(30))+1   '
print @sql
```

```
declare @VInt1 int,@VInt2 int,@VInt3 int,@selectsql
varchar(max),@updatesql varchar(max);
select @VInt1=1,@VInt2=2,@VInt3=3;
set @selectsql='select '+ltrim(@VInt1)+' * '+ltrim(@VInt2)+' *
'+ltrim(@VInt3)+'';  --查询
set @updatesql='update 表名set 字段名= '+ltrim(@VInt1)+' *
'+ltrim(@VInt2)+' * '+ltrim(@VInt3)+' ';
--update 表名set 字段名='1*2*3'
print(@updatesql)
```

Dense_rank()函数使用

```
/*
声明一个变量当做Table来处理.
*/

declare @Table table (Province varchar(4),City varchar(4))
insert into @Table
select '广东','广州' union all
select '广东','深圳' union all
select '广东','汕头' union all
select '海南','海口' union all
select '海南','三亚' union al l
select null,'香港' union all
select null,'澳门'



;with cte as   --获取排序后的顺序
(
   select
            Dense_rank() OVER(order by case when Province is not null
then 'a' else 'z' end
            ,ISNULL(Province,City) asc) as DR,Province,City

         from @table
),cte_RowNumber as   ---加入分组后的顺序
(
  select
       *,ROW_NUMBER() OVER(PARTITION BY ISNULL(Province,City) order by
getdate()) AS RN
from cte
)

select Case when RN=1 then ltrim(DR) else '' end as Num,
Province,City
from cte_RowNumber
order by DR

/*
```

数据库项目文档

```
Num                    Province City
---------------------- -------- ----
1                      广东     广州
                       广东     深圳
                       广东     汕头
2                      海南     海口
                       海南     三亚
3                      NULL     澳门
4                      NULL     香港
*/

--select 序号= case
--        when pid = 1 then
--         ltrim(row_number() over(partition by pid order by rid))
--        else
--         ''
--      end,
--      省,
--      市
--  from (select row_number() over(order by getdate()) as rid,
--              row_number() over(partition by isnull(省, 市) order by
getdate()) as pid,
--              *
--         from @t) aa
-- order by rid
```

表添加默认值的做法

```
create table student(
    stuId int identity(1,1) primary key,--想在这里加入主键，这个我已经写好了！
    stuName varchar(10) unique not null,--想在这里加入唯一约束，怎么写？
    stuAge int Check(stuAge between 18 and 100),--想在这里加入检查约束怎么
写？大于=18，小于=100就好。
    stuAddress varchar(20) default 'china',--想在这里加入默认约束怎么？
    stuDel varchar(20)
)
```

FullJoin 选择不同表的字段

```
/*Full Join*/
SELECT ISNULL(Tbl1.ID,Tbl2.Id) as Id,ISNULL(Tbl1.Name,Tbl2.Name) as
Name,
       ISNULL(Tbl1.Amount,0)+ISNULL(Tbl2.Amount,0) as Amount
FROM Tbl1 full join
    Tbl2 on Tbl1.id=Tbl2.ID
```

Not in 和 Not Exist 是否存在

```
/*not in写行*/
select * from tableB
union all
(
 select * from tableA where serverid not in(select serverid from tableB)
)

select * from tableB
union all
(
 select * from tableA where not exists (
 select * from tableB where serverid=tableA.serverid)
)

/*not exist  的左外连接也行也行*/

select * from tableB
union all
select * from tableA a
where not exists(select 1 from tableB where serverid=a.serverid and
driverid=a.driverid)
```

字符串截取问题截取*前面和后面的问题

```
/*
请问一下表A：
ID comment
```

数据库项目文档

```
1 12*13
2 124*3
3 ASD*4R
4 23D
.....

规则，若有*，截取*前面的数值，若没有，只需带出原有的数值
*/
if object_id(N'A','U') is not null drop table A
go
create table A
(
 id int not null,
 comment varchar(50) null
)
go
insert into A
select 1,'12*13' union all
select 2,'124*13' union all
select 3,'ASD*4r' union all
select 4,'23D' union all


select id,(case when charindex('*',comment)>0
              then left(comment,charindex('*',comment)-1)
         else comment end/*上边少个end*/) comment
from a

--select ID , substring(comment ,1,charindex('*',comment )) from A

    ---select charindex('*',comment) from a




    declare @t table (colname varchar(27))
    insert into @t
    select 'abc 公司&李海涛/135648490034'

    select left(colname ,charindex('&',colname )-1) colname from @t
    select substring(colname ,1,charindex('&',colname )-1) colname from
```

@t

字符串截取问题 replicate 函数使用

/*
现有sql200数据库
数据库A1
表Nu
数据类型全部是varchar

SN NAME BIANMA
1 张三50
2 李四60

用什么数据库命令可以直接讲上述表格显示成为满位数值
如：
SN NAME BIANMA
1 张三00050
2 李四00060

求助大家，最好把命令格式直接给我。谢谢

以指定的次数重复字符串值。

*/

```
declare @table table (SN int,NAME varchar(4),BIANMA int)
insert into @table
select 1,'张三',50 union all
select 2,'李四',60

select SN,name,BIANMA=
right(replicate('0',5)+ltrim(BIANMA),5) from @table
/*
SN          name BIANMA
----------- ---- -----------
```

```
1          张三  00050
2          李四  00060
*/


/*
REPLICATE (Transact-SQL)
SQL Server 2008 R2 其他版本 SQL Server 2008 SQL Server 2005

以指定的次数重复字符串值。

Transact-SQL 语法约定

语法
----------------------------------------------------------------------
-----------

复制

REPLICATE ( string_expression ,integer_expression )
参数
----------------------------------------------------------------------
-----------

string_expression
字符串或二进制数据类型的表达式。string_expression 可以是字符或二进制数据。

 注意
如果string_expression 的类型不是varchar(max) 或nvarchar(max)，则REPLICATE
将截断返回值，截断长度为8,000 字节。若要返回大于8,000 字节的值，则必须将
string_expression 显式转换为适当的大值数据类型。


integer_expression
任何整数类型的表达式，包括bigint。如果integer_expression 为负，则返回NULL。


    */一
```

字符串截取问题 如何用逗号把他们分割出来然后每个号码分别插入到表@phone="15117172362,13965699989,150121"

```
/*
标题：简单数据拆分(version 2.0)
作者：爱新觉罗.毓华(十八年风雨,守得冰山雪莲花开)
时间：-05-07
地点：重庆航天职业学院
描述：

有表tb，如下：
id          value
----------- -----------
1           aa,bb
2           aaa,bbb,ccc
欲按id,分拆value列，分拆后结果如下：
id          value
----------- --------
1           aa
1           bb
2           aaa
2           bbb
2           ccc
*/

--1. 旧的解决方法(sql server 2000)

create table tb(id int,value varchar(30))
insert into tb values(1,'aa,bb')
insert into tb values(2,'aaa,bbb,ccc')
go

--方法.使用临时表完成
SELECT TOP 8000 id = IDENTITY(int, 1, 1) INTO # FROM syscolumns a,
syscolumns b

SELECT A.id, value = SUBSTRING(A.[value], B.id, CHARINDEX(',', A.[value]
+ ',', B.id) - B.id)
FROM tb A, # B
```

```
WHERE SUBSTRING(',' + A.[value], B.id, 1) = ','

DROP TABLE #

--方法.如果数据量小，可不使用临时表
select a.id , value = substring(a.value , b.number , charindex(',' ,
a.value + ',' , b.number) - b.number)
from tb a join master..spt_values  b
on b.type='p' and b.number between 1 and len(a.value)
where substring(',' + a.value , b.number , 1) = ','

--2. 新的解决方法(sql server 2005)
create table tb(id int,value varchar(30))
insert into tb values(1,'aa,bb')
insert into tb values(2,'aaa,bbb,ccc')
go

--方法.使用xml完成
SELECT A.id, B.value FROM
(
  SELECT id, [value] = CONVERT(xml,'<root><v>' + REPLACE([value], ',',
'</v><v>') + '</v></root>') FROM tb
) A OUTER APPLY
(
  SELECT value = N.v.value('.', 'varchar(100)') FROM
A.[value].nodes('/root/v') N(v)
) B

--方法.使用CTE完成
;with tt as
(select id,[value]=cast(left([value],charindex(',',[value]+',')-1) as
nvarchar(100)),Split=cast(stuff([value]+',',1,charindex(',',[value]+'
,'),'') as nvarchar(100)) from tb
union all
select id,[value]=cast(left(Split,charindex(',',Split)-1) as
nvarchar(100)),Split= cast(stuff(Split,1,charindex(',',Split),'') as
nvarchar(100)) from tt where split>''
)
select id,[value] from tt order by id option (MAXRECURSION 0)
```

数据库项目文档

```
DROP TABLE tb

/*
id          value
----------- ------------------------------
1           aa
1           bb
2           aaa
2           bbb
2           ccc

(5 行受影响)
   */
declare @phone varchar(50)
set @phone='15117172362,13965699989,15012110123'
create table tb(phone varchar(15))
insert into tb
select
substring(@phone,number,charindex(',',@phone+',',number+1)-number)
from master..spt_values
where type='p' and number<=len(@phone) and
substring(@phone,number,1)<>',' and substring(','+@phone,number,1)=','
go
select * from tb
/*
phone
---------------
15117172362
13965699989
15012110123

(3 行受影响)

*/
go
   drop table tb


--那我补充下：

create table tb(phone varchar(15))
```

```
go

declare @str varchar(100)
set @str = '15117172362,13965699989,15012110123'

insert into tb
select substring(@str,number,charindex(',',@str+',',number)-number)
from master..spt_values
where [type] = 'p' and number between 1 and len(@str)
    and substring(','+@str,number,1) = ','

select *
from tb

drop table tb

/************

phone
---------------
15117172362
13965699989
15012110123

(3 行受影响)


insert into tb_queue
 select null,@spno,@linkid,
      substring(@str,number,charindex(',',@str+',',number)-number),
    @msg,null,@status,@userid,@pwd,@sid,
    @InsertTime,@SendTime,@STSRPTTime,@SP_SendTime,@SP_STSRPTTime,
    @Fix_BeginTime,@Fix_EndTime,@sendlevel
from master..spt_values
where [type] = 'p' and number between 1 and len(@str)
     and substring(','+@str,number,1) = ','
```

两张表一起更新的做法

```
----两张表一起更新的语句
update lzhu set zaiwang=A.zaiwang
from kkkkk as A
where kkkkk.haoma=lzhu.haoma
```

根据 id 由小到大排列，自动把 jilu 中为 null 的值补为上一条记录的值 结果为，求用 sql 怎么实现

```
select id,
       (case
         when jilu is not null then
          jilu
         else
          (select top 1 jilu
             from tb
            where id < t.id
              and jilu is not null
            order by id desc)
       end) jilu
      from tb t
```

我想查询表 A 和表 B.表 A 查询全部数据 表 B 只查询最新插入 tongji 的一条记录

```
  /*
表A id name fenshu zongfen
表B id tongji

2张表没有主外键，只是单纯的张表

我想查询表A和表B

表A查询全部数据表B只查询最新插入tongji的一条记录

select A.*, b.tongji from A
```

```
left join B on A.Id=b.Id

我只想查B表的第一条记录，其他的不查，应该怎么修改，或者用其他方法？

谢谢！
*/
select A.*, b.tongji
from A
   left join
  (select * from B t where not exists (select 1 from B where tongji = t.tongji
and id > t.id))B
   on A.Id=b.Id
```

7.BI 要复习的知识点如下.

维度建模

维度建模（dimensional modeling）是数据仓库建设中的一种数据建模方法。Kimball 最先提出这一概念。其最简单的描述就是，按照事实表，维表来构建数据仓库，数据集市。这种方法的最被人广泛知晓的名字就是星型模式（Star-schema）。实体关系(E-R)建模通常用于为单位的所有进程创建一个复杂的模型。这种方法已被证实在创建高效的联机事务处理 (OLTP) 系统方面很有效。相反，维度建模针对零散的业务进程创建个别的模型。例如，销售信息可以创建为一个模型，库存可以创建为另一个模型，而客户帐户也可以创建为另一个模型。每个模型捕获事实数据表中的事实，以及那些事实在链接到事实数据表的维度表中的特性。由这些排列产生的架构称为星型架构或雪花型架构，已被证实在数据仓库设计中很有效。　　维度建模将信息组织到结构中，这些结构通常对应于分析者希望对数据仓库数据使用的查询方法。1999 年第三季度西北地区的食品销售额是多少？表示使用三个维度（产品、地理、时间）指定要汇总的信息。星型模式之所以广泛被使用，在于针对各个维作了大量的预处理，如按照维进行预先的统计、分类、排序等。通过这些预处理，能够极大的提升数据仓库的处理能力。特别是针对 3NF 的建模方法，星型模式在性能上占据明显的优势。　　同时，维度建模法的另外一个优点是，维度建模非常直观，紧紧围绕着业务模型，可以直观的反映出业务模型中的业务问题。不需要经过特别的抽象处理，即可以完成维度建模。这一点也是维度建模的优势。　　但是，维度建模法的缺点也是非常明显的，由于在构建星型模式之前需要进行大量的数据预处理，因此会导致大量的数据处理工作。而且，当业务发生变化，需要重新进行维度的定义时，往往需要重新进行维度数据的预处理。而在这些与处理过程中，往往会导致大量的数据冗余。　　另外一个维度建模法的缺点就是，如果只是依靠单纯的维度建模，不能保证数据来源的一致性和准确性，而且在数据仓库的底层，不是特别适用于维度建模的方法。

星型模型和雪花模型的区别

星形模式是一种多维的数据关系，它由一个事实表（Fact Table）和一组维表（Dimension Table）组成。每个维表都有一个维作为主键，所有这些维的主键组合成事实表的主键。事实表的非主键属性称为事实（Fact），它们一般都是数值或其他可以进行计算的数据；而维大都是文字、时间等类型的数据，按这种方式组织好数据我们就可以按照不同的维（事实表主键的部分或全部）来对这些事实数据进行求和（summary）、求平均（average）、计数（count）、百分比（percent）的聚集计算，甚至可以做20～80分析。这样就可以从不同的角度数字来分析业务主题的情况。）

在多维分析的商业智能解决方案中，根据事实表和维度表的关系，又可将常见的模型分为星型模型和雪花型模型。在设计逻辑型数据的模型的时候，就应考虑数据是按照星型模型还是雪花型模型进行组织。

当所有维表都直接连接到“ 事实表”上时，整个图解就像星星一样，故将该模型称为星型模型，如图 2 。

星型架构是一种非正规化的结构，多维数据集的每一个维度都直接与事实表相连接，不存在渐变维度，所以数据有一定的冗余，如在地域维度表中，存在国家 A 省 B 的城市 C 以及国家 A 省 B 的城市 D 两条记录，那么国家 A 和省 B 的信息分别存储了两次，即存在冗余。

当有一个或多个维表没有直接连接到事实表上，而是通过其他维表连接到事实表上时，其图解就像多个雪花连接在一起，故称雪花模型。雪花模型是对星型模型的扩展。它对星型模型的维表进一步层次化，原有的各维表可能被扩展为小的事实表，形成一些局部的 " 层次 " 区域，这些被分解的表都连接到主维度表而不是事实表。如图 2-3，将地域维表又分解为国家，省份，城市等维表。它的优点是 ： 通过最大限度地减少数据存储量以及联合较小的维表来改善查询性能。雪花型结构去除了数据冗余

星型模型因为数据的冗余所以很多统计查询不需要做外部的连接，因此一般情况下效率比雪花型模型要高。星型结构不用考虑很多正规化的因素，设计与实现都比较简单。 雪花型模型由于去除了冗余，有些统计就需要通过表的联接才能产生，所以效率不一定有星型模型高。正规化也是一种比较复杂的过程，相应的数据库结构设计、数据的 ETL、以及后期的维护都要复杂一些。因此在冗余可以接受的前提下，实际运用中星型模型使用更多，也更有效率。

数据仓库与数据集市

数据仓库，英文名称为Data Warehouse，可简写为DW。

数据仓库之父Bill Inmon在1991年出版的“Building the Data Warehouse”一书中所提出的定义被广泛接受——数据仓库（Data Warehouse）是一个面向主题的（Subject Oriented）、集成的（Integrated）、相对稳定的（Non-Volatile）、反映历史变化（Time Variant）的数据集合，用于支持管理决策(Decision Making Support)。

◆面向主题：操作型数据库的数据组织面向事务处理任务，各个业务系统之间各自分离，而数据仓库中的数据是按照一定的主题域进行组织的。

◆集成的：数据仓库中的数据是在对原有分散的数据库数据抽取、清理的基础上经过系统加工、汇总和整理得到的，必须消除源数据中的不一致性，以保证数据仓库内的信息是关于整个企业的一致的全局信息。

◆相对稳定的：数据仓库的数据主要供企业决策分析之用，所涉及的数据操作主要是数据查询，一旦某个数据进入数据仓库以后，一般情况下将被长期保留，也就是数据仓库中一般有大量的查询操作，但修改和删除操作很少，通常只需要定期的加载、刷新。

◆反映历史变化：数据仓库中的数据通常包含历史信息，系统记录了企业从过去某一时点（如开始应用数据仓库的时点）到目前的各个阶段的信息，通过这些信息，可以对企业的发展历程和未来趋势做出定量分析和预测。

数据仓库是一个过程而不是一个项目。

数据仓库系统是一个信息提供平台，他从业务处理系统获得数据，主要以星型模型和雪花模型进行数据组织，并为用户提供各种手段从数据中获取信息和知识。

从功能结构化分，数据仓库系统至少应该包含数据获取（Data Acquisition）、数据存储（Data Storage）、数据访问（Data Access）三个关键部分。

企业数据仓库的建设，是以现有企业业务系统和大量业务数据的积累为基础。数据仓库不是静态的概念，只有把信息及时交给需要这些信息的使用者，供他们做出改善其业务经营的决策，信息才能发挥作用，信息才有意义。而把信息加以整理归纳和重组，并及时提供给相应的管理决策人员，是数据仓库的根本任务。因此，从产业界的角度看，数据仓库建设是一个工程，是一个过程。

整个数据仓库系统是一个包含四个层次的体系结构，具体由下图表示。

数据库项目文档

数据仓库系统体系结构

•数据源：是数据仓库系统的基础，是整个系统的数据源泉。通常包括企业内部信息和外部信息。内部信息包括存放于RDBMS中的各种业务处理数据和各类文档数据。外部信息包括各类法律法规、市场信息和竞争对手的信息等等；

•数据的存储与管理：是整个数据仓库系统的核心。数据仓库的真正关键是数据的存储和管理。数据仓库的组织管理方式决定了它有别于传统数据库，同时也决定了其对外部数据的表现形式。要决定采用什么产品和技术来建立数据仓库的核心，则需要从数据仓库的技术特点着手分析。针对现有各业务系统的数据，进行抽取、清理，并有效集成，按照主题进行组织。数据仓库按照数据的覆盖范围可以分为企业级数据仓库和部门级数据仓库（通常称为数据集市）。

•OLAP(联机分析处理)服务器：对分析需要的数据进行有效集成，按多维模型予以组织，以便进行多角度、多层次的分析，并发现趋势。其具体实现可以分为：ROLAP（关系型在线分析处理）、MOLAP（多维在线分析处理）和HOLAP（混合型线上分析处理）。ROLAP基本数据和聚合数据均存放在RDBMS之中；MOLAP基本数据和聚合数据均存放于多维数据库中；HOLAP 基本数据存放于RDBMS之中，聚合数据存放于多维数据库中。

•前端工具：主要包括各种报表工具、查询工具、数据分析工具、数据挖掘工具以数据挖掘及各种基于数据仓库或数据集市的应用开发工具。其中数据分析工具主要针对OLAP 服务器，报表工具、数据挖掘工具主要针对数据仓库。

目前，数据仓库一词尚没有一个统一的定义，著名的数据仓库专家W.H.Inmon在其著作《Building the Data Warehouse》一书中给予如下描述：数据仓库（Data Warehouse）是一个面向主题的（Subject Oriented）、集成的（Integrate）、相对稳定的（Non-Volatile）、反映历史变化（Time Variant）的数据集合，用于支持管理决策。对于数据仓库的概念我们可以从两个层次予以理解，首先，数据仓库用于支持决策，面向分析型数据处理，它不同于企业现有的操作型数据库；其次，数据仓库是对多个异构的数据源有效集成，集成后按照主题进行了重组，并包含历史数据，而且存放在数据仓库中的数据一般不再修改。多维

数据仓库的组成

数据仓库数据库

是整个数据仓库环境的核心，是数据存放的地方和提供对数据检索的支持。相对于操纵型数据库来说其突出的特点是对海量数据的支持和快速的检索技术。

数据抽取工具

把数据从各种各样的存储方式中拿出来，进行必要的转化、整理，再存放到数据仓库内。对

各种不同数据存储方式的访问能力是数据抽取工具的关键，应能生成COBOL程序、MVS作业控制语言（JCL）、UNIX脚本、和SQL语句等，以访问不同的数据。数据转换都包括，删除对决策应用没有意义的数据段；转换到统一的数据名称和定义；计算统计和衍生数据；给缺值数据赋给缺省值；把不同的数据定义方式统一。

元数据

元数据是描述数据仓库内数据的结构和建立方法的数据。可将其按用途的不同分为两类，技术元数据和商业元数据。

技术元数据是数据仓库的设计和管理人员用于开发和日常管理数据仓库是用的数据。包括：数据源信息；数据转换的描述；数据仓库内对象和数据结构的定义；数据清理和数据更新时用的规则；源数据到目的数据的映射；用户访问权限，数据备份历史记录，数据导入历史记录，信息发布历史记录等。

商业元数据从商业业务的角度描述了数据仓库中的数据。包括：业务主题的描述，包含的数据、查询、报表；

元数据为访问数据仓库提供了一个信息目录（informationdirectory），这个目录全面描述了数据仓库中都有什么数据、这些数据怎么得到的、和怎么访问这些数据。是数据仓库运行和维护的中心，数据仓库服务器利用他来存贮和更新数据，用户通过他来了解和访问数据。

访问工具

为用户访问数据仓库提供手段。有数据查询和报表工具；应用开发工具；管理信息系统（EIS）工具；在线分析（OLAP）工具；数据挖掘工具。

数据集市（DataMarts）

为了特定的应用目的或应用范围，而从数据仓库中独立出来的一部分数据，也可称为部门数据或主题数据（subjectarea）。在数据仓库的实施过程中往往可以从一个部门的数据集市着手，以后再用几个数据集市组成一个完整的数据仓库。需要注意的就是再实施不同的数据集市时，同一含义的字段定义一定要相容，这样再以后实施数据仓库时才不会造成大麻烦。

数据仓库管理：安全和特权管理；跟踪数据的更新；数据质量检查；管理和更新元数据；审计和报告数据仓库的使用和状态；删除数据；复制、分割和分发数据；备份和恢复；存储管理。

信息发布系统：把数据仓库中的数据或其他相关的数据发送给不同的地点或用户。基于Web的信息发布系统是对付多用户访问的最有效方法。

设计数据仓库的九个步骤

数据库项目文档

1)选择合适的主题（所要解决问题的领域）

2)明确定义fact表

3)确定和确认维

4)choosing the facts

5)计算并存储fact表中的衍生数据段

6)roundingoutthedimensiontables

7)choosingthedurationofthedatabase

8)theneedtotrackslowlychangingdimensions

9)确定查询优先级和查询模式。

技术上

硬件平台：数据仓库的硬盘容量通常要是操作数据库硬盘容量的2-3倍。通常大型机具有更可靠的性能和和稳定性，也容易与历史遗留的系统结合在一起；而PC服务器或UNIX服务器更加灵活，容易操作和提供动态生成查询请求进行查询的能力。选择硬件平台时要考虑的问题：是否提供并行的I/O吞吐？对多CPU的支持能力如何？

数据仓库DBMS：他的存储大数据量的能力、查询的性能、和对并行处理的支持如何。

网络结构：数据仓库的实施在那部分网络段上会产生大量的数据通信，需不需要对网络结构进行改进。

实现上

建立数据仓库的步骤

1)收集和分析业务需求

2)建立数据模型和数据仓库的物理设计

3)定义数据源

4) 选择数据仓库技术和平台

5) 从操作型数据库中抽取、净化、和转换数据到数据仓库

6) 选择访问和报表工具

7) 选择数据库连接软件

8) 选择数据分析和数据展示软件

9) 更新数据仓库

数据抽取、清理、转换、和移植

1) 数据转换工具要能从各种不同的数据源中读取数据。

2) 支持平面文件、索引文件、和legacyDBMS。

3) 能以不同类型数据源为输入整合数据。

4) 具有规范的数据访问接口

5) 最好具有从数据字典中读取数据的能力

6) 工具生成的代码必须是在开发环境中可维护的

7) 能只抽取满足指定条件的数据，和源数据的指定部分

8) 能在抽取中进行数据类型转换和字符集转换

9) 能在抽取的过程中计算生成衍生字段

10) 能让数据仓库管理系统自动调用以定期进行数据抽取工作，或能将结果生成平面文件

11) 必须对软件供应商的生命力和产品支持能力进行仔细评估

主要数据抽取工具供应商：
Prismsolutions.Carleton'sPASSPORT.InformationBuildersInc.'s

EDA/SQL.SASInstituteInc.

数据库项目文档

数据仓库带来了什么

每一家公司都有自己的数据。并且，许多公司在计算机系统中储存有大量的数据，记录着企业购买、销售、生产过程中的大量信息和客户的信息。通常这些数据都储存在许多不同的地方。

使用数据仓库之后，企业将所有收集来的信息存放在一个唯一的地方——数据仓库。仓库中的数据按照一定的方式组织，从而使得信息容易存取并且有使用价值。

目前，已经开发出一些专门的软件工具，使数据仓库的过程实现可以半自动化，帮助企业将数据倒入数据仓库，并使用那些已经存入仓库的数据。

数据仓库给组织带来了巨大的变化。数据仓库的建立给企业带来了一些新的工作流程，其他的流程也因此而改变。

数据仓库为企业带来了一些“以数据为基础的知识”，它们主要应用于对市场战略的评价，和为企业发现新的市场商机，同时，也用来控制库存、检查生产方法和定义客户群。

每一家公司都有自己的数据。数据仓库将企业的数据按照特定的方式组织，从而产生新的商业知识，并为企业的运作带来新的视角。

为何要建立数据仓库

计算机发展的早期，人们已经提出了建立数据仓库的构想。“数据仓库”一词最早是在1900年，由Bill Inmon先生提出的，其描述如下：数据仓库是为支持企业决策而特别设计和建立的数据集合。

企业建立数据仓库是为了填补现有数据存储形式已经不能满足信息分析的需要。数据仓库理论中的一个核心理念就是：事务型数据和决策支持型数据的处理性能不同。

企业在它们的事务操作收集数据。在企业运作过程中：随着定货、销售记录的进行，这些事务型数据也连续的产生。为了引入数据，我们必须优化事务型数据库。

处理决策支持型数据时，一些问题经常会被提出：哪类客户会购买哪类产品？促销后销售额会变化多少？价格变化后或者商店地址变化后销售额又会变化多少呢？在某一段时间内，相对其他产品来说哪类产品特别容易卖呢？哪些客户增加了他们的购买额？哪些客户又削减了他们的购买额呢？

事务型数据库可以为这些问题作出解答，但是它所给出的答案往往并不能让人十分满意。在运用有限的计算机资源时常常存在着竞争。在增加新信息的时候我们需要事务型数据库是空闲的。而在解答一系列具体的有关信息分析的问题的时候，系统处理新数据的有效性又会被大大降低。另一个问题就在于事务型数据总是在动态的变化之中的。决策支持型处理需要相对稳定的数据，

从而问题都能得到一致连续的解答。

数据仓库的解决方法包括：将决策支持型数据处理从事务型数据处理中分离出来。数据按照一定的周期（通常在每晚或者每周末），从事务型数据库中导入决策支持型数据库——既“数据仓库”。数据仓库是按回答企业某方面的问题来分“主题”组织数据的，这是最有效的数据组织方式。

数据仓库和数据集市

有关决策支持型数据库的数据集市是面向企业中的某个部门或是项目小组的。一些专家顾问将数据集市的建造描述为建立数据仓库全过程中的一步。首先，一个储存企业全部信息的数据仓库被创建，其中，数据均具备有组织的、一致的、不变的格式。数据集市随后被创立，其目的是为不同部门提供他们所需要的那部分信息。数据仓库聚集了所有详细的信息，而数据集市中的数据则是针对用户们的特定需求总结而出的。

而另外一些专家则认为数据集市的建立并不需要首先建立一个数据仓库。在这个模型中，数据直接由事务型数据库转入数据集市中。一个公司可能建立有多个数据集市，而彼此之间毫无联系。

这种不在建立数据仓库的基础上创建数据集市的方式会更便宜、更快速，因为它的规模更加易于管理。

第二种观点的缺陷在于无法实现最初创建数据仓库的最主要的目的——将企业所有的数据统一为一致的格式。现有的事务处理系统的数据往往是不一致、冗余的。如果首先建立起一个全公司范围的数据仓库，组织就能够获得一个统一关于企业的活动和客户的知识库。如果先建立起一个个独立的数据集市，那么数据仓库的诸多优势都能够得以实现，但是企业远远无法做到对数据的一致的储存。

代理键的设置.

KDT#81 事实表中的代理键

在数据仓库的建模中，代理键通常是建立在维度表中的。那么在事实表中如何呢？在建立逻辑模型时，事实表中显然是没必要建立代理键的，但是到了物理模型，在某些特定的情况下是可以考虑建立代理键。

代理键一般是无意义的整型值，做为维度表的主键，它的分配过程一般是顺序的。代理键可以很好的隔离源系统的数据变化，对数据仓库中的查询性能也能起到很好的作用。在事实表中，主键一般定义为维度外键的子集，通常几个维度外键即可实现主键的功能。这样的情况下，在事实表中定义代理键是起不了什么作用的。

在下面举例的一些情况，事实表也可以考虑使用代理键。

1．有时，企业的业务规则允许重复的记录存在。当然，这时我们需要尽力去源系统中找各种可能使记录能唯一标识的字段，如时间戳等，来使数据记录能唯一性。如果最终的结果是不得不接受这种重复记录时，可以采用代理键来实现数据记录的唯一性。

2．某些时候，增加代理键可以使ETL的处理方便很多。尤其是，处理Update的方式采用先Insert再Delete的时候。这时增加代理键可以保证策略能使用，并且能提高加载性能。如果要操作的事实表中是复合主键时，采用该策略时都可以建立代理键。

3．另一个ETL的需求是，当加载工作暂停时，使用顺序分配的代理键可以很容易的判断出继续工作的位置。

总结来说，代理键对于维度表来说是一个非常好的选择。对事实表来说，在数据迁移时使用代理键会起到非常大的作用。

代理键和OLTP系统
大凡代理键都是为了处理数据仓库系统的缓慢变化维而设计的，那OLTP系统有无必要使用代理键呢？
1：首先我们搞清楚什么是代理键
通常数据库设计时会选择一个候选键作为主键，而主键通常是人可识别并且带一定商业含义的域，我们称之为自然键；
代理键就是代替那个自然键作为主键，而代理键通常是没有含义的流水号。
2：使用自然键可能存在的问题
1）必须首先确定自然键，才能输入数据。比如，某个业务系统需要给A客户分配一个代码，当A客户提交资料后，首先要给定一个客户代码，然后才能录入系统；
2）自然键的改变比较困难，比如像深交所的股票代码，从四位升级到六位，改变相关资料是非常大的工作量；
3：使用代理键的好处和坏处
1）更改变得非常容易，比如客户表，因为客户代码不再是主键，所以很容易更改，不需要更改引用它的表；
2）新增数据时可以后确定客户代码，因为代理键作为主键了。比如我们还是考虑2：中的业务场景，某个业务系统需要给A客户建立资料，分配客户代码，我们可以先将该客户录入

系统，目前客户代码是空的，然后我们提交相关的资料给相关部门，相关部门根据业务资料时候合适决定是否通过，如通过则手工或者自动分配一个客户代码，然后返回给提交人

员，否则驳回申请资料或者关闭申请。
3）和其他表的关联上没有那么直观。

个人意见，是否使用代理键见仁见智的事了！

渐变维度处理【缓慢变化维】-多层次保存历史记录讲解

数据建模，我们经常会碰到使用维度表。为了能够追踪历史，我们通常有两种方法：一种是用时间戳（起始生效日期和结束日期），一种是使用是否当前有效的字段标记。这里使用这些字段的目的，在于采集事实表时，能根据这些标记查找对应的维度记录 ID（通常使用代理键，不要使用自然键，并且更新时应该先更新维度表，在更新事实表）。

但是我们经常会碰到存在层次关系的维度，在这些维度记录中通常通过 ID 作为层次关系的根据。如下图所示的维度物理表：

	代理键	分公司ID	分公司名称	部门ID	部门	人员ID	人员名	有效开始时间	有效结束时间
1	1	1001	深圳中地	1001	市场部	2001	人员甲	2010-12-02 00:00:00.000	NULL
2	2	1001	深圳中地	1001	市场部	2002	人员乙	2010-12-02 00:00:00.000	NULL
3	3	1001	深圳中地	1002	研发部	2003	人员A	2010-12-02 00:00:00.000	NULL

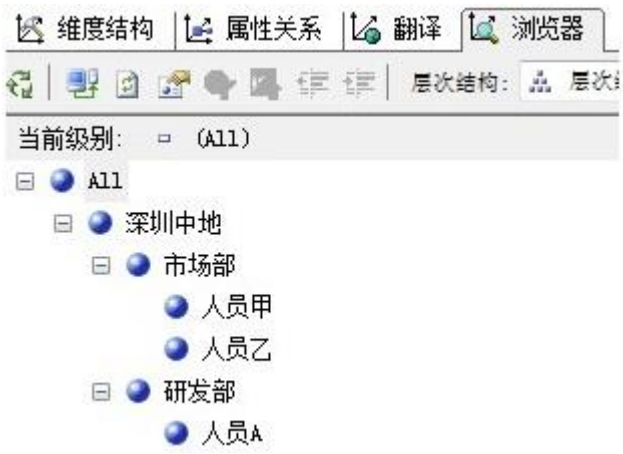
设置好层次关系。

属性

	名称	用法	类型	键列
	部门ID	Regular	Regular	Integer
	代理键	Key	Regular	Integer
	分公司ID	Regular	Regular	Integer
	人员ID	Regular	Regular	Integer
	若要创建新属性，请将列拖至			

选中属性列表中的属性，设置部门，分公司，人员对应的显示名称（选中上面属性行，在属性窗口的“NameColumn”属性进行设置）。

查看结果如下：



加入现在 2010-12-3 日，市场部的名字更改成了“销售部”。更通过渐变维度更新，最将记录 1、2 的有效结束日期更改为 2010-12-3 日，并添加两条新的记录。如下图所示：

	代理键	分公司ID	分公司名称	部门ID	部门	人员ID	人员名	有效开始时间	有效结束时间
1	1	1001	深圳中地	1001	市场部	2001	人员甲	2010-12-02 00:00:00.000	2010-12-03 00:00:00.00
2	2	1001	深圳中地	1001	市场部	2002	人员乙	2010-12-02 00:00:00.000	2010-12-03 00:00:00.00
3	3	1001	深圳中地	1002	研发部	2003	人员A	2010-12-02 00:00:00.000	NULL
4	4	1001	深圳中地	1001	销售部	2001	人员甲	2010-12-03 00:00:00.000	NULL
5	5	1001	深圳中地	1001	销售部	2002	人员乙	2010-12-03 00:00:00.000	NULL

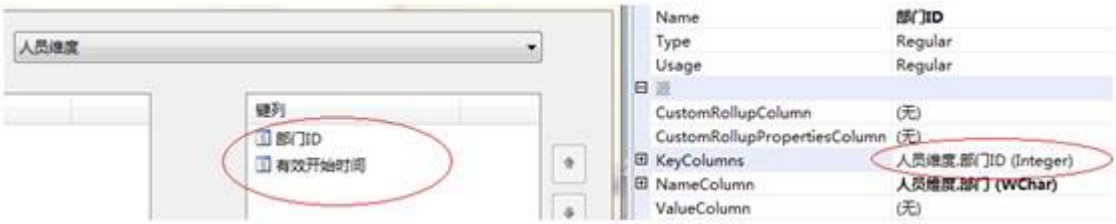
注意：记录 3 没有做任何改动，如果 2010-12-3 日进行事实表更新，则事实表中关联的 ID 为 3、4、5 而不再是 1、2、3。

如果我们此时重新更新维度，更新到 SSAS 服务器，则会报错“OLAP 存储引擎中存在错误：处理时找到重复的属性键：表：“dbo_人员维度”，列：“部门 ID”，值：“1001”。该属性为“部门 ID”。”

因为更新维度时，1001 对应另个名字“市场部”“销售部”。我们必须对老的记录和新记录的数据进行“隔离”。

选中属性列表中的部门 ID 设置“键列”（设置属性窗口中的 KeyColumns），我们看到只有“部门 ID”，把“有效开始时间”也放进来变成组合键列，并且

人员也进行同样的设置。最顶级别的“分公司 ID”则用“分公司 ID”和“分公司名称”做为组合键。



更新到服务器后，可以看到如下的层次结构：

有些人会说，当前“市场部”已经不要了，怎么还出来了。其实这个没关系，因为我们查看事实表的时候有个时间维度，选定一个日期关联的事实表，而同一天的这些事实表不可能同时有关联“市场部”又有关联“销售部”的。

总之：最高级别的“分公司”的键列为“分公司 ID”+“分公司名称”；其他的键列为“ID”+“有效其实日期”。

大家踊跃发言，批评指正！！

为了能够追踪历史，我们通常有两种方法：

1.时间戳（起始生效日期和结束日期）；

2.使用是否当前有效的字段标记。

这里使用这些字段的目的，在于采集事实表时，能根据这些标记查找对应的维度记录 ID（通常使用代理键，不要使用自然键，并且更新时应该先更新维度表，在更新事实表）。

但是我们经常会碰到存在层次关系的维度，在这些维度记录中通常通过 ID 作为层次关系的根据。如下图所示的维度物理表：

设置好层次关系。

选中属性列表中的属性，设置部门，分公司，人员对应的显示名称（选中上面属性行，在属性窗口的“NameColumn”属性进行设置）。

查看结果如下：

加入现在 2010-12-3 日，市场部的名字更改成了“销售部”。更通过渐变维度更新，最将记录 1、2 的有效结束日期更改为 2010-12-3 日，并添加两条新的记录。如下图所示：

注意：记录 3 没有做任何改动，如果 2010-12-3 日进行事实表更新，则事实表中关联的 ID 为 3、4、5 而不再是 1、2、3。

如果我们此时重新更新维度，更新到 SSAS 服务器，则会报错“OLAP 存储引擎中存在错误: 处理时找到重复的属性键: 表:“dbo_人员维度”，列:“部门 ID”，值:“1001”。该属性为“部门 ID”。”

因为更新维度时，1001 对应另个名字“市场部”“销售部”。我们必须对老的记录和新记录的数据进行“隔离”。

选中属性列表中的部门 ID 设置“键列”(设置属性窗口中的 KeyColumns)，我们看到只有“部门 ID”，把“有效开始时间”也放进来变成组合键列，并且人员也进行同样的设置。最顶级别的“分公司 ID”则用“分公司 ID”和“分公司名称”做为组合键。

更新到服务器后，可以看到如下的层次结构：

有些人会说，当前“市场部”已经不要了，怎么还出来了。其实这个没关系，因为我们查看事实表的时候有个时间维度，选定一个日期关联的事实表，而同一天的这些事实表不可能同时有关联“市场部”又有关联“销售部”的。

总之：最高级别的“分公司”的键列为“分公司 ID”+“分公司名称”；其他的键列为“ID”+“有效其实日期”。

怎么使用 MDX 查询历史数据：

这里的时间戳，只是为了事实表关联维度表的时候，能根据这个时间戳（有效标志）找到当时应该对应的维度记录行。实际查询的时候，应该根据时间查出事实表，然后事实表表记录自动关联起历史中的维度

本文来自 CSDN 博客，转载请标明出处：

http://blog.csdn.net/jzdzhiyun/archive/2010/12/03/6052993.aspx

为了能够追踪历史，我们通常有两种方法：

1.时间戳（起始生效日期和结束日期）；

2.使用是否当前有效的字段标记。

这里使用这些字段的目的，在于采集事实表时，能根据这些标记查找对应的维度记录 ID（通常
理键，不要使用自然键，并且更新时应该先更新维度表，在更新事实表）。

但是我们经常会碰到存在层次关系的维度，在这些维度记录中通常通过 ID 作为层次关系的根
图所示的维度物理表：

	代理键	分公司ID	分公司名称	部门ID	部门	人员ID	人员名	有效开始时间	有效结束时间
1	1	1001	深圳中地	1001	市场部	2001	人员甲	2010-12-02 00:00:00.000	NULL
2	2	1001	深圳中地	1001	市场部	2002	人员乙	2010-12-02 00:00:00.000	NULL
3	3	1001	深圳中地	1002	研发部	2003	人员A	2010-12-02 00:00:00.000	NULL

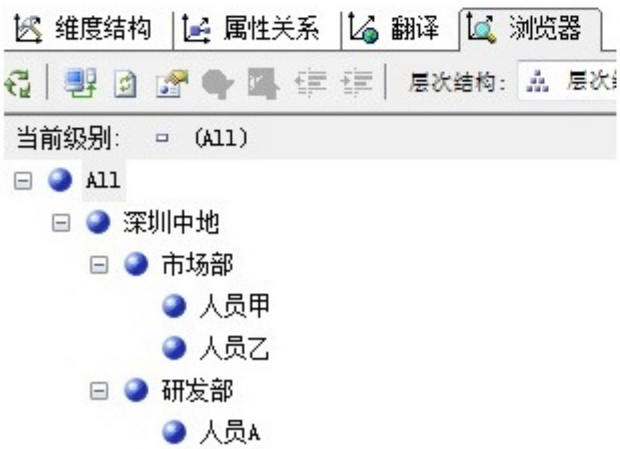
设置好层次关系。

属性

	名称	用法	类型	键列
	部门ID	Regular	Regular	Integer
	代理键	Key	Regular	Integer
	分公司ID	Regular	Regular	Integer
	人员ID	Regular	Regular	Integer

选中属性列表中的属性，设置部门，分公司，人员对应的显示名称（选中上面属性行，在属性窗口的“Name
属性进行设置）。

查看结果如下：



加入现在 2010-12-3 日，市场部的名字更改成了“销售部”。更通过渐变维度更新，最将记录 1、2 的有

数据库项目文档

---

http://files.cnblogs.com/emmy/testfu.rar

只能在 vs2008 下才能新建维度属性关系。在 vs2005 上没有这个页签。

缓慢变化的维表：

员工代理键	生效日期	失效日期	员工 ID	员工姓名	所属部门
1	20010101	20091231	A	张三	营业部
2	20100101	20991231	A	张三	市场部

事实表：

员工代理键	数据所属日期	事实数据
1	20050101	100.00
2	20100501	220.00

归纳：

1、缓慢变化维可以认为是维度的关键属性发生变化时，根据变化时间将数据进行拆分并独立 ID（代理键）化（个人观点描述，大体是这样）

2、维度中必须包含生命周期（有效时间段）和虚拟的 ID（代理键），这样才能有效区分数据的变化

3、事实表中用维表 ID（代理键）进行外键匹配，保证按照时间统计可区分维度属性的变化

4、事实表尽量包含一个数据所属日期，以保证数据是可逆的，也就是说当事实表损坏的时候可利用数据所属日期重新匹配维表(数据所属日期 between （生命周期）)

5.6 读后感：

1.数据仓库的 维表设计：代理键，用户 ID，用户 name，生效日期，失效日期

事实表设计：代理键，数据所属日期

2.如果维表发生变化，首先在维表添加一条新数据，生效日期，并把旧数据的失效日期改为新数据的生效日期。

3.当有新的事实数据产生时，我们先用

select 维表代理键 from 维表

where 用户 ID＝XX

and

事实表的数据所属日期 between 维表的生效日期 and 维表的失效日期 。

把得到的代理键插入到事实表的新一条数据中。

4.在建 ssas 的 cube 时，用事实表的代理键 关联 维表的代理键 ，事实表的日期 关联 日期维度的日期 ID

5.cube 建立完成后，查询数据一定要选日期维度。

最后：ssas 的 cube 只能做关联 代理键，拿代理键对应的数据，对数据的处理 还得在数据仓库设计和 ETL 这一段解决.....

报表，分析，挖掘，BI 的三个层次.

经过几年的积累，大部分中大型的企事业单位已经建立了比较完善的CRM、ERP、OA等基础信息化系统。这些系统的统一特点都是：通过业务人员或者用户的操作，最终对数据库进行增加、修改、删除等操作。上述系统可统一称为OLTP（Online Transaction Process，在线事务处理），指的就是系统运行了一段时间以后，必然帮助企事业单位收集大量的历史数据。但是，在数据库中分散、独立存在的大量数据对于业务人员来说，只是一些无法看懂的天书。业务人员所需要的是信息，是他们能够看懂、理解并从中受益的抽象信息。此时，如何把数据转化为

信息，使得业务人员（包括管理者）能够充分掌握、利用这些信息，并且辅助决策，就是商业智能主要解决的问题。

如何把数据库中存在的数据转变为业务人员需要的信息？大部分的答案是报表系统。简单说，报表系统已经可以称作是BI了，它是BI的低端实现。

现在国外的企业，大部分已经进入了中端BI，叫做数据分析。有一些企业已经开始进入高端BI，叫做数据挖掘。而我国的企业，目前大部分还停留在报表阶段。

数据报表不可取代

传统的报表系统技术上已经相当成熟，大家熟悉的Excel、水晶报表、Reporting Service等都已经被广泛使用。但是，随着数据的增多，需求的提高，传统报表系统面临的挑战也越来越多。

1．数据太多，信息太少

密密麻麻的表格堆砌了大量数据，到底有多少业务人员仔细看每一个数据？到底这些数据代表了什么信息、什么趋势？级别越高的领导，越需要简明的信息。如果我是董事长，我可能只需要一句话：目前我们的情况是好、中还是差？

2．难以交互分析、了解各种组合

定制好的报表过于死板。例如，我们可以在一张表中列出不同地区、不同产品的销量，另一张表中列出不同地区、不同年龄段顾客的销量。但是，这两张表无法回答诸如“华北地区中青年顾客购买数码相机类型产品的情况”等问题。业务问题经常需要多个角度的交互分析。

3．难以挖掘出潜在的规则

报表系统列出的往往是表面上的数据信息，但是海量数据深处潜在含有哪些规则呢？什么客户对我们价值最大，产品之间相互关联的程度如何？越是深层的规则，对于决策支持的价值越大，但是，也越难挖掘出来。

4．难以追溯历史，数据形成孤岛

业务系统很多，数据存在于不同地方。太旧的数据（例如一年前的数据）往往被业务系统备份出去，导致宏观分析、长期历史分析难度很大。

因此，随着时代的发展，传统报表系统已经不能满足日益增长的业务需求了，企业期待着新的技术。数据分析和数据挖掘的时代正在来临。值得注意的是，数据分析和数据挖掘系统的目的是带给我们更多的决策支持价值，并不是取代数据报表。报表系统依然有其不可取代的优势，并且将会长期与数据分析、挖掘系统一起并存下去。

八维以上的数据分析

如果说OLTP侧重于对数据库进行增加、修改、删除等日常事务操作，OLAP（Online Analytics Process，在线分析系统）则侧重于针对宏观问题，全面分析数据，获得有价值的信息。

为了达到OLAP的目的，传统的关系型数据库已经不够了，需要一种新的技术叫做多维数据库。

多维数据库的概念并不复杂。举一个例子，我们想描述2003年4月份可乐在北部地区销售额10万元时，牵扯到几个角度：时间、产品、地区。这些叫做维度。至于销售额，叫做度量值。当然，还有成本、利润等。

如图2，每个维度分别代表了时间、产品和地区，立方体上的单元代表了度量值。进一步，维度可以分为不同的层次

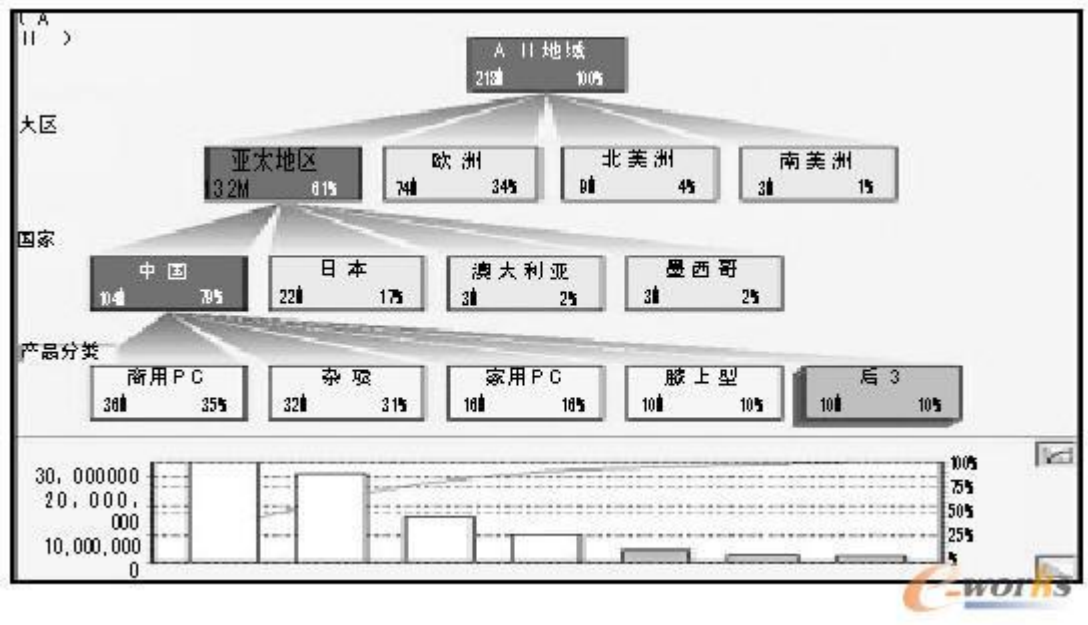


图 1 某案例中对销售额的解析及当前产品的分类

数据库项目文档

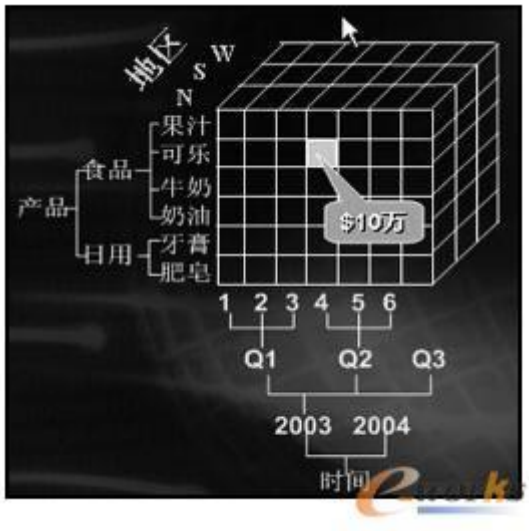


图 2 使用多维数据分析的案例

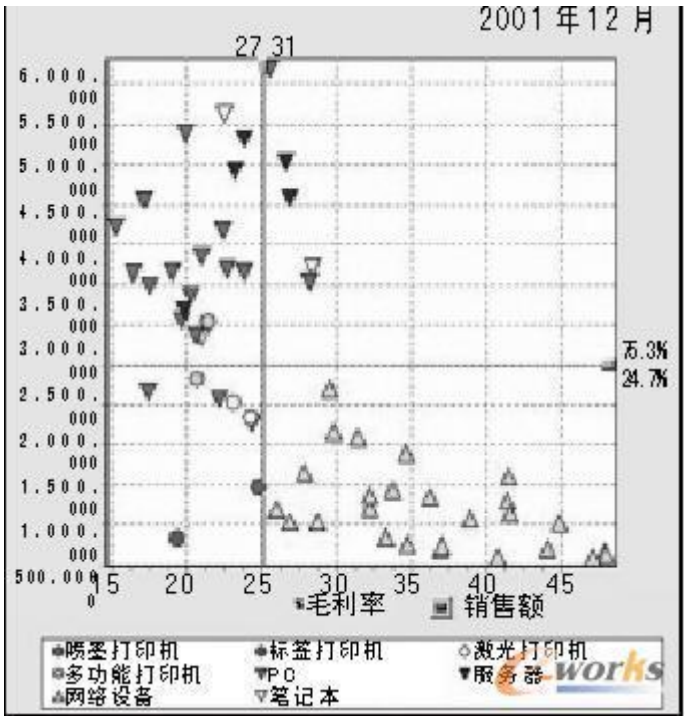


图 3 某案例的数据分析投影图

除了时间、产品和地区，我们还可以有很多维度，例如客户的性别、职业、销售部门、促销方式等等。实际上，使用中的多维数据库可能是一个8维或者15维的立方体。

虽然结构上15维的立方体很复杂，但是概念上非常简单

数据分析系统的总体架构分为四个部分：源系统、数据仓库、多维数据库、客户端。

• 源系统：包括现有的所有OLTP系统，搭建BI系统并不需要更改现有系统。

• 数据仓库：数据大集中，通过数据抽取，把数据从源系统源源不断地抽取出来，可能每天一次，或者每3个小时一次，当然是自动的。数据仓库依然建立在关系型数据库上，往往符合叫做“星型结构”的模型。

• 多维数据库：数据仓库的数据经过多维建模，形成了立方体结构。每一个立方体描述了一个业务主题，例如销售、库存或者财务。

• 客户端：好的客户端软件可以把多维立方体中的信息丰富多彩地展现给用户。

数据分析案例：

在实际的案例中，我们利用Oracle 9i搭建了数据仓库，Microsoft Analysis Service 2000搭建了多维数据库，ProClarity 6.0 作为客户端分析软件。

分解树好像一个组织图。分解树在回答以下问题时很有效：

• 在指定的产品组内，哪种产品有最高的销售额？

• 在特定的产品种类内，各种产品间的销售额分布如何？

• 哪个销售人员完成了最高百分比的销售额？

在图1中，可以对PC机在各个地域的销售额和所占百分比一目了然。任意一层分解树都可以根据不同维度随意展开。在该分解树中，在大区这一层是按国家展开，在国家这一层是按产品分类展开。

投影图（图3）使用散点图的格式，显示两个或三个度量值之间的关系。数据点的集中预示两个变量之间存在强的相关关系，而稀疏分布的数据点可能显示不明显的关系。

投影图很适合分析大量的数据。在显示因果关系方面有明显效果，比如例外的数据点就可以考虑进一步研究，因为它们落在“正常”的点群范围之外。

数据挖掘看穿你的需求

广义上说，任何从数据库中挖掘信息的过程都叫做数据挖掘。从这点看来，数据挖掘就是BI。但从技术术语上说，数据挖掘（Data Mining）特指的是：源数据经过清洗和转换等成为适合于挖掘的数据集。数据挖掘在这种具有固定形式的数据集上完成知识的提炼，最后以合适的知识模式用于进一步分析决策工作。从这种狭义的观点上，我们可以定义：数据挖掘是从特定形式的数据集中提炼知识的过程。数据挖掘往往针对特定的数据、特定的问题，选择一种或者多种挖掘

算法，找到数据下面隐藏的规律，这些规律往往被用来预测、支持决策。

关联销售案例：

美国的超市有这样的系统：当你采购了一车商品结账时，售货员小姐扫描完了你的产品后，计算机上会显示出一些信息，然后售货员会友好地问你：我们有一种一次性纸杯正在促销，位于F6货架上，您要购买吗？

这句话决不是一般的促销。因为计算机系统早就算好了，如果你的购物车中有餐巾纸、大瓶可乐和沙拉，则86%的可能性你要买一次性纸杯。结果是，你说，啊，谢谢你，我刚才一直没找到纸杯。

这不是什么神奇的科学算命，而是利用数据挖掘中的关联规则算法实现的系统。

每天，新的销售数据会进入挖掘模型，与过去N天的历史数据一起，被挖掘模型处理，得到当前最有价值的关联规则。同样的算法，分析网上书店的销售业绩，计算机可以发现产品之间的关联以及关联的强弱。

数据报表、数据分析、数据挖掘是BI的三个层面。我们相信未来几年的趋势是：越来越多的企业在数据报表的基础上，会进入数据分析与数据挖掘的领域。商业智能所带来的决策支持功能，会给我们带来越来越明显的效益。

BI 中的报表与业务系统中的报表有什么区别？

1． BI 中的报表与业务系统中的报表有什么区别？

在BI实施过程中经常会遇到，做的报表业务系统也存在，而且在实施的测试阶段是拿业务系统的报表来验证BI中的报表。

首先我要说明的是，看业务需求是什么样的，如果是向领导汇报（周报、日报等），这种做法是十分正确的，但如果是分析决策或改进管理为目的的话，那么这种做法就是掩耳盗铃。一句话无用功。大家想想，分析准确性的保证是什么，一定是数据的准确并有效。业务数据包含了各种真实发生的情况数据， 当然是准确的，但并不是全部有效地。 所以BI中才叫ETL过程，而不叫数据迁移。

因此BI中的报表与业务系统中的报表第一个区别就是：BI中的报表数据对分析来说是准确、有

效地。

第二个区别就是BI教会你如何看报表。听起来让人好笑，那我就详细解释一下吧，就拿资产负债表来说，不同的岗位的人，看到该表后得出的结论可能存在不同，因为他们出发点、关注点不同，所以会得出不同的结论。因此三人行必有我师，在BI系统中，通过报表帮助可以教会你从各个角度看报表。从而提高你的管理水平，一致达到准确决策的目的。（千万不要告诉我业务系统也可以做到，当然可以做到，IT无所不能，只要有规则，但你能实施成功吗？去该业务系统。）

第三个区别就是知识体系。普通à优秀à卓越，差距在哪里呢，知识的宏观把控是一项非常重要的内容吧。因此，BI中的报表与报表间，已经形成了知识体系，达到了宏观的高度，即形成了知识体系。

举例说明：我们在看多张报表时候，一定有一定顺序的看，当一张数据有问题，我们会找相关的报表查找。这个过程就是一个小小的知识体系。

第三个区别BI报表具有BI功能

…… 还有很多地方不同，这里就不一一介绍了。

2． BI真的能达到你说的那样的效果吗？

这个属于一个售前的问题，在BI售前过程中，参与的都是高级领导，他们已经经历过ERP的“折磨”了（上ERP等于（），不上ERP等于（）。请自行填空吧）。

如何让客户放心，呵呵，对于一个优秀的BI人员来说，现场制作报表，是传统的、有效地方法。这里我就不多做介绍了。我介绍的是其它方法，

方法一：分析模型现场演示

以BO或 share point中的excel service为例，都提供了相应的工具，非常方便制作，可以事先做好例子。然后现场输入数据（也可体现输入），进行模拟演示。起到沙盘的作用。一般客户都会被高度吸引。

方法二：定制演示

这里指的不是拖拖拽拽。而是个性化定制，最好的例子就是KPI的个性化定制演示。这也是客户最关心的，可以随意方便的定制（度量和纬度范围）。

方法三：系统集成

不要等客户提出这个问题，而是主动提出。我在做售前过程中，基本每次都被问到（客户经常是一个系统一个用户名一个密码，对老大们来说，不断地在不同系统间输入用户名和密码是种折磨），后来我就主动说了。

…..还有很多方法，在这里就不一一介绍了，如大家有什么更好的方法，请多交流。（俺QQ：51612395,请注明：BI）

BI 项目实施随笔

目前BI对大多说中、大型企业来说绝对是一个热门话题。商业智能很多企业通过购买行业BI软件或定制BI项目开发等形式实施，无论如何实施，企业最终想通过BI来

1. 整合企业数据

2. 满足大量的报表需求。

3. 解决管理与决策问题。

那么BI是如何满足以上需求的呢？简单来说：

1. 通过ETL整合企业数据

2. 通过BI的前端展现，来完成大量报表的展现。

3. 通过BI的个性化功能与工具，来满足管理与决策的问题。例如：BI 门户，指标设计器、预警系统、杜邦分析模型、ABC客户模型。

数据库项目文档

BI毕竟是一个新的领域，很多人都是从ERP顾问转过来的，无论是售前人员、销售人员、实施人员都会经常需要回答以下客户提出的问题：

1. BI 到底能给我们带来什么？

2. BI 中的报表与业务系统中的报表有什么区别？

3. BI真的能达到你说的那样的效果吗？

4. BI的挖掘功能是怎么挖的，我能否把我需要的因素都加入进去？

5. BI的预测功能是如何预测的？（注意：预测和挖掘可是有区别的啊）

6. BI应该如何选型？

以上几个问题是我在做ERP的BI售前、实施过程中经常被问及到的问题。 那么我就针对以上问题一一解释一下，我个人的观点。

在回答问题前我用三句话一下我的BI世界观：

1. BI的定义：

BI 是把公司从成立到现在，积累的数据、业务知识、管理知识，以及企业未来的发展方向，进行业务梳理与整合。通过BI技术，形成大量的报表、管理和决策工具，从而形成一套完整的知识体系。进而满足从最高决策人到部门管理人员，到业务人员分析与应用的信息系统。

2. BI的实施原则

把知识体系工具化和展现化，BI实施重点在于分析模型、分析工具、平台建设为重点，展现报表只是其中的一部分工作（平台已经预置了大量需求的报表，做的只是特殊需求的展现），当然不同项目会有不同的工作重点。

（目前国内BI实施的公司来说，

a. 对政府的。基本是以报表开发为主，根本谈不上BI的平台化，所以这些企业基本上是以报表起家的公司。全是国内企业。）

b. 对企业的。

除央企外，基本上都是购买ERP公司的BI软件。

央企：典型代表中国电信、银行，基本上是按业务、部门来一个一个的以项目形式走。

3. BI的效果

公司有什么样的管理水平，就有什么样的BI。BI是一项长期的工作，是随着公司与市场的变化而变化的，不同的发展阶段应该有不同的工作重点。

OK,接下来，我就用我的BI世界观来回答以上提出的6个问题。

1. BI到底给我们带来了什么？

有过BI售前经验的人，一定在售前的第一个环节就回答了这个问题。那是通过“非常绚丽的”、“实用的工具”图片来给客户进行演示。

看似客户已经了解了这个问题，但在实施的过程中、以致项目上线后，都会有客户不断地提出这个问题，为什么呢？（引用小沈阳，呵呵！ 本人可是地道、正宗的沈阳人，大家可以叫我Eric-Zheng）。答案很简单，由于售前所描绘的理想与现实的差距，以及现实中都是报表，决策分析很少体现所导致给问题被反复提到。

我是这么回答的：

BI系统是分析决策的工具。

分析：来源于数据，展现于报表。目前报表已经给出了分析所需的数据内容，至于分析，在报表的百科全书（BO系统中存在的一项非常重要功能，但绝大多数人没有利用起来。以后我会专门介绍该功能，congos也有该功能）中已经给出了各种分析。

决策：首先明确，要让计算机帮你决策，必须把您的决策思想程序化，这样系统才能根据数据进行决策。（这样的回答是把问题推回给客户了，因为客户头痛的正是这点，他们也没有形成。因此，上BI前最好是由咨询公司规划一下。）我们是软件实施厂商，不是咨询公司。应该是您把您的决策思想告诉我们，然后我们通过计算机自动化来实现。

BI 挖掘的体现方式

4． 为什么看不到挖掘功能？挖应该如何体现。

BI中最吸引人的地方就是挖啊。但往往项目结束，客户也没看到如何挖的。（这就是现实）。什么神经网络、聚类分析、线性回归、决策树…… 都哪里应用了啊。

销售预测分析，库存预测分析、预算预测分析怎么都没了。

我的回答是：

杜邦分析，ABC客户模型都是挖的啊。你提到的销售预测等，请告诉我你平时工作中是如何挖的，即挖的思想，然后我给你做挖。退一步说，如果客户真的提出了挖的思想，那么先不要着急去做，首先第一步是衡量提出的参数是否可量化。第二步是验证和确认的过程，第三步调整方案的确认，毕竟挖是随着时间不断调整的。最后才是实施。

把握一句：BI无所不能，只要告诉我规则。（呵呵，废话一句，其实对IT来说都是一样，技术已经不是问题了。）

5． BI的预测功能是如何预测的？

首先预测和挖掘是有区别的，字典含义：

a． 挖掘：把蕴藏着的力量或积极性等发挥出来：挖掘潜力。

b. 预测：预先推测或测定。

挖掘是来自置身的，预测不但需要来自自身还需要来自外部。

BI中关键的一点是，不但要通过环比、同比来自身对比，更需要同行业、甚至整个市

场环境的同比与环比。

所以谈到预测就必须存在行业数据或参照数据（可以是目标值）。然后再谈预测。

综上，必须建立目标体系模型才能实现真正的预测。

6. BI应该如何选型？

现在很多企业都面临着同样的问题，首先当然是钱。呵呵，其实这个问题和我们平时买东西是一个道理。

A. BI市场体会

首先想看你预算是多少，一分钱一分货，当然你也可以去北京的“秀水”或上海的“南京街”，也可以去沈阳的“五爱市场”（家乡的，很大啊，东西很多，便宜，欢迎来选购）。 “秀水”的好东西比正货使用起来不差多少。对于BI来说一个道理，现在的BI市场很乱，有点沾边的公司都敢上。即使一流的ERP企业也一样，没有标准产品，先投到标再说。更何况还存在像XX、XX这种国内大的集成商啊。（本人置身其中，深有感受）。说句心里话，对于BI来说，你可以说他100元也行，说100万也行。这就是市场。

B. 公司情况

这里不是说的公司规模。而指的IT技术实力，我碰到过这样的公司，他们IT部门200多人，自我感觉技术很强，居然自己要从建模到前端web展现的软件（模仿congos,bo）。哈哈，如果你要能模仿出来，哪些大公司几十年的努力岂不都白费了。所以请把你的关注点放在业务上而不是技术上。技术买就可以了。

C. 需求与要求

根据公司发展情况，确定需求与要求。如果公司情况达不到，及时你上最好的BI效果也等于0.

D． 实施工具体会

国际品牌、国内高端品牌、一般品牌 产品陈词补齐。

使用满足需求才是硬道理。高端品牌虽然性能出色，但在中国式报表上还是国内的土工具实用。但如果真是上真正的BI的话，相信我，选择国际品牌。SAP等贵有贵的道理，国内的高端品牌更本不是一个数量级的。

注意：如果你是政府项目，请选择国内。因为……（你自己知道的）

举个具体例子：现在国内数一数二的ERP大厂商，都是和国外的品牌OEM的。但其中也有自己想独立做的，例如某公司收购了一家BI公司，然后进行准备替代，不知道什么时候能真正替代。本人拭目以待，期待国货走向世界级。

E． 服务

BI不是产品贵，而是服务贵。你买手机只是工具，信号好坏的看电信的。

数据仓库和数据挖掘技术的实现

数据仓库的重要概念

维： 待补

数据立方体：

当用户观察某一事物的角度不同时，围绕该事物会产生多个观察角度，也就是产生了多维。数据仓库中的多种维交点，就是数据仓库用户所需要观察的事务。例如：数据立方体中由客户、产品、时间三个维所构成的立方体表示哪些客户、在什么地方购买了那些产品。三个维的交点就是所购买产品数量或者价格等事务，也就是立方体的顶点。数据仓库的立方体实际上是一个包含用户需要观察数据的集合体，它提供企业所感兴趣的商业事务。在这里最重要的是购买的产品与价格等信息，这些信息构成了立方体的粒度，即维交叉时所导致的细节等级，如果某个客户购买了某种商品，结果就是一个基本粒度或原子事务。立方体作为基本事务的聚合，是一种适合通过SQL或者他接口进行查询的完整数据结构。一般而言立方体可以转换成星型模型，星型模型也可

以转换成立方体。

在数据仓库的实际应用中，高层的数据聚合存储采用立方体处理，效率较高。而细节为基础，维变化的上卷聚合采用星型模型处理效率更高并且更灵活，

立方体也称为多维数据集。超过三维的立方体称为超立方体，或超维数据集。

聚集 ：

聚集或者聚合是指收集了基本事务数据的结构。在一个立方体中包含了很多层次，这些层次可以向用户提供某一层次的概括数据。

数据仓库的未来发展

1.基于关系对象数据库的数据仓库

关系对象数据库的初相使数据仓库设计人员有能力将对象引入数据仓库环境中。

关系数据库参考：
http://wiki.ccw.com.cn/%E5%85%B3%E7%B3%BB%E6%95%B0%E6%8D%AE%E5%BA%93

http://baike.baidu.com/view/68348.htm

2.网络影响

3.操作型数据库仓库，能以一种可以接受的标准对数据仓库进行操作。这些标准包括可预知性、可利用性和可访问性。

4.Web应用技术代理

数据仓库的应用

2类用户：信息使用者，知识挖掘者

数据库项目文档

数据仓库挖掘者数据分析的4个过程

1.概括分析：首先对数据仓库中的数据外部特征进行分析，确保数据的完整性和准确性，评价是否有充分的样本数据进行数据抽取、建模与分类处理。

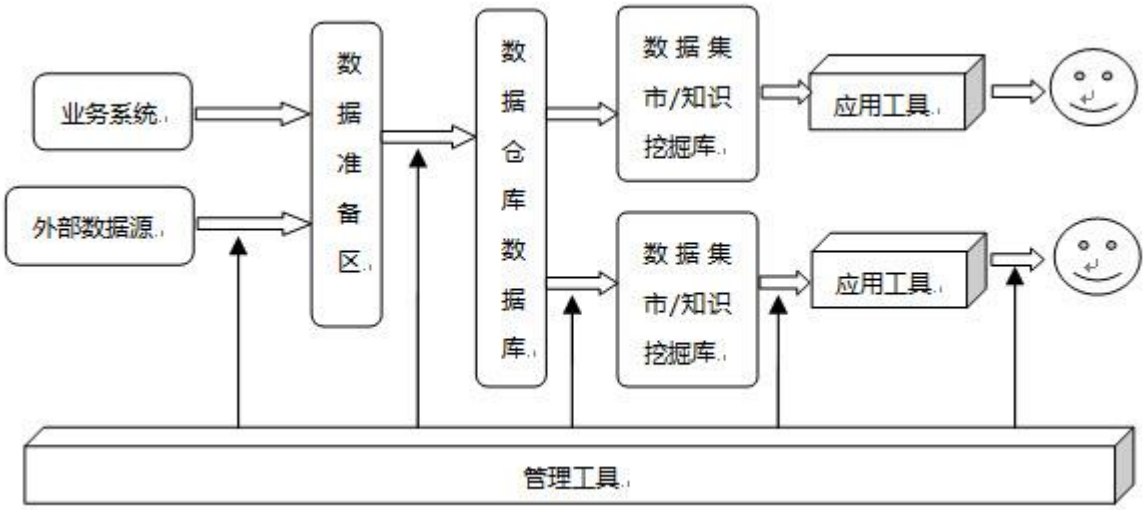
概括分析内容可能有:常来采购的客户性别比例多大？共有多少客户？经常进行采购的客户数量比例情况如何？客户的平均采购标准是多少？

2.数据抽取：根据知识挖掘的需要对数据仓库中的数据进行抽取。按照数据分析的目的，对这些数据进行组织，然后将组织好的数据送入数据集市或者知识挖掘库中。

3.建模分析：使用数据仓库的核心工作，是开发一种用于描述客户、产品或销售商模型的过程。在完成建模分析后,就可以利用所建的模型对数据仓库中的实体与模型的关联程度进行分类分析。

4.分类处理：根据挖掘出来的知识对数据库中的所有数据进行分类。分类的目的是针对不同的事务采取正确的对策，这也是知识挖掘者挖掘知识的最终目的。

数据仓库系统结构



8.SQLServer 的调优工具和性能优化工具及执行计划详解

提高 SQL 性能的方法

根本原因通常在于数据库设计和访问它的查询这些技术可用于提高基于 SQL Server? 的应用程序的性能或改善其可伸缩性

内嵌视图与临时表
某些时候，查询需要将数据与其他一些可能只能通过执行 GROUP BY 然后执行标准查询才能收集的数据进行联接。例如，如果要查询最新五个定单的有关信息，您首先需要知道是哪些定单。这可以使用返回定单 ID 的 SQL 查询来检索。此数据就会存储在临时表（这是一个常用技术）中，然后与 Products 表进行联接，以返回这些定单售出的产品数量:SQL 语句会创建一个临时表，将数据插入该表中，将其他数据与该表进行联接，然后除去该临时表。这会导致此查询进行大量 I/O 操作，因此，可以重新编写查询，使用内嵌视图取代临时表。内嵌视图只是一个可以联接到 FROM 子句中的查询。所以，您不用在 tempdb 中的临时表上耗费大量 I/O 和磁盘访问，而可以使用内嵌视图得到同样的结果。此工具可能会被过度使用。LEFT JOIN 消耗的资源非常之多，因为它们包含与 NULL（不存在）数据匹配的数据。在某些情况下，这是不可避免的，但是代价可能非常高。LEFT JOIN 比 INNER JOIN 消耗资源更多，所以如果您可以重新编写查询以使得该查询不使用任何 LEFT JOIN，则会得到非常可观的回报。加快使用 LEFT JOIN 的查询速度的一项技术涉及创建一个 TABLE 数据类型，插入第一个表（LEFT JOIN 左侧的表）中的所有行，然后使用第二个表中的值更新 TABLE 数据类型。此技术是一个两步的过程，但与标准的 LEFT JOIN 相比，可以节省大量时间。一个很好的规则是尝试各种不同的技术并记录每种技术所需的时间，直到获得用于您的应用程序的执行性能最佳的查询。

另一个提高效率的技巧是使用 DISTINCT 关键字查找数据行的单独报表，来代替使用 GROUP BY 子句。在这种情况下，使用 DISTINCT 关键字的 SQL 效率更高。请在需要计算聚合函数（SUM、COUNT、MAX 等）的情况下再使用 GROUP BY。另外，如果您的查询总是自己返回一个唯一的行，则不要使用 DISTINCT 关键字。在这种情况下，DISTINCT 关键字只会增加系统开销。

SqlServer 性能调优综述

我始终认为，一个系统的性能的提高，不单单是试运行或者维护阶段的性能调优的任务，也不单单是开发阶段的事情，而是在整个软件生命周期都需要注意，进行有效工作才能达到的。所

数据库项目文档

以我希望按照软件生命周期的不同阶段来总结数据库性能优化相关的注意事项。

一、 分析阶段

一般来说，在系统分析阶段往往有太多需要关注的地方，系统各种功能性、可用性、可靠性、安全性需求往往吸引了我们大部分的注意力，但是，我们必须注意，性能是很重要的非功能性需求，必须根据系统的特点确定其实时性需求、响应时间的需求、硬件的配置等。最好能有各种需求的量化的指标。

另一方面，在分析阶段应该根据各种需求区分出系统的类型，大的方面，区分是OLTP（联机事务处理系统）和OLAP（联机分析处理系统）。

二、 设计阶段

设计阶段可以说是以后系统性能的关键阶段，在这个阶段，有一个关系到以后几乎所有性能调优的过程—数据库设计。

在数据库设计完成后，可以进行初步的索引设计，好的索引设计可以指导编码阶段写出高效率的代码，为整个系统的性能打下良好的基础。

以下是性能要求设计阶段需要注意的：

1、 数据库逻辑设计的规范化

数据库逻辑设计的规范化就是我们一般所说的范式，我们可以这样来简单理解范式：

第1规范：没有重复的组或多值的列，这是数据库设计的最低要求。

第2规范：每个非关键字段必须依赖于主关键字，不能依赖于一个组合式主关键字的某些组成部分。消除部分依赖，大部分情况下，数据库设计都应该达到第二范式。

第3规范：一个非关键字段不能依赖于另一个非关键字段。消除传递依赖，达到第三范式应该是系统中大部分表的要求，除非一些特殊作用的表。

更高的范式要求这里就不再作介绍了，个人认为，如果全部达到第二范式，大部分达到第三范式，系统会产生较少的列和较多的表，因而减少了数据冗余，也利于性能的提高。

2、 合理的冗余

完全按照规范化设计的系统几乎是不可能的，除非系统特别的小，在规范化设计后，有计划地加入冗余是必要的。

冗余可以是冗余数据库、冗余表或者冗余字段，不同粒度的冗余可以起到不同的作用。

冗余可以是为了编程方便而增加，也可以是为了性能的提高而增加。从性能角度来说，冗余数据库可以分散数据库压力，冗余表可以分散数据量大的表的并发压力，也可以加快特殊查询的速度，冗余字段可以有效减少数据库表的连接，提高效率。

3、 主键的设计

主键是必要的，SQL SERVER的主键同时是一个唯一索引，而且在实际应用中，我们往往选择最小的键组合作为主键，所以主键往往适合作为表的聚集索引。聚集索引对查询的影响是比较大的，这个在下面索引的叙述。

在有多个键的表，主键的选择也比较重要，一般选择总的长度小的键，小的键的比较速度快，同时小的键可以使主键的B树结构的层次更少。

主键的选择还要注意组合主键的字段次序，对于组合主键来说，不同的字段次序的主键的性能差别可能会很大，一般应该选择重复率低、单独或者组合查询可能性大的字段放在前面。

4、 外键的设计

外键作为数据库对象，很多人认为麻烦而不用，实际上，外键在大部分情况下是很有用的，理由是：

外键是最高效的一致性维护方法，数据库的一致性要求，依次可以用外键、CHECK约束、规则约束、触发器、客户端程序，一般认为，离数据越近的方法效率越高。

谨慎使用级联删除和级联更新，级联删除和级联更新作为SQL SERVER 2000当年的新功能，在2005作了保留，应该有其可用之处。我这里说的谨慎，是因为级联删除和级联更新有些突破了传统的关于外键的定义，功能有点太过强大，使用前必须确定自己已经把握好其功能范围，否则，级联删除和级联更新可能让你的数据莫名其妙的被修改或者丢失。从性能看级联删除和级联更新是比其他方法更高效的方法。

5、 字段的设计

字段是数据库最基本的单位，其设计对性能的影响是很大的。需要注意如下：

A、数据类型尽量用数字型，数字型的比较比字符型的快很多。

B、 数据类型尽量小，这里的尽量小是指在满足可以预见的未来需求的前提下的。

C、 尽量不要允许NULL，除非必要，可以用NOT NULL+DEFAULT代替。

D、少用TEXT和IMAGE，二进制字段的读写是比较慢的，而且，读取的方法也不多，大部分情况下最好不用。

E、 自增字段要慎用，不利于数据迁移。

6、 数据库物理存储和环境的设计

在设计阶段，可以对数据库的物理存储、操作系统环境、网络环境进行必要的设计，使得我们的系统在将来能适应比较多的用户并发和比较大的数据量。

这里需要注意文件组的作用，适用文件组可以有效把I/O操作分散到不同的物理硬盘，提高并发能力。

7、 系统设计

整个系统的设计特别是系统结构设计对性能是有很大影响的，对于一般的OLTP系统，可以选择C/S结构、三层的C/S结构等，不同的系统结构其性能的关键也有所不同。

系统设计阶段应该归纳一些业务逻辑放在数据库编程实现，数据库编程包括数据库存储过程、触发器和函数。用数据库编程实现业务逻辑的好处是减少网络流量并可更充分利用数据库的预编译和缓存功能。

8、 索引的设计

在设计阶段，可以根据功能和性能的需求进行初步的索引设计，这里需要根据预计的数据量和查询来设计索引，可能与将来实际使用的时候会有所区别。

关于索引的选择，应改主意：

A、 根据数据量决定哪些表需要增加索引，数据量小的可以只有主键。

B、 根据使用频率决定哪些字段需要建立索引，选择经常作为连接条件、筛选条件、聚合查询、排序的字段作为索引的候选字段。

C、 把经常一起出现的字段组合在一起，组成组合索引，组合索引的字段顺序与主键一样，也需要把最常用的字段放在前面，把重复率低的字段放在前面。

D、 一个表不要加太多索引，因为索引影响插入和更新的速度。

三、 编码阶段

编码阶段是本文的重点，因为在设计确定的情况下，编码的质量几乎决定了整个系统的质量。

编码阶段首先是需要所有程序员有性能意识，也就是在实现功能同时有考虑性能的思想，数据库是能进行集合运算的工具，我们应该尽量的利用这个工具，所谓集合运算实际是批量运算，就是尽量减少在客户端进行大数据量的循环操作，而用SQL语句或者存储过程代替。关于思想和意识，很难说得很清楚，需要在编程过程中来体会。

下面罗列一些编程阶段需要注意的事项：

1、 只返回需要的数据

返回数据到客户端至少需要数据库提取数据、网络传输数据、客户端接收数据以及客户端处理数据等环节，如果返回不需要的数据，就会增加服务器、网络和客户端的无效劳动，其害处是显而易见的，避免这类事件需要注意：

A、横向来看，不要写SELECT *的语句，而是选择你需要的字段。

B、 纵向来看，合理写WHERE子句，不要写没有WHERE的SQL语句。

C、 注意SELECT INTO后的WHERE子句，因为SELECT INTO把数据插入到临时表，这个过程会锁定一些系统表，如果这个WHERE子句返回的数据过多或者速度太慢，会造成系统表长期锁定，诸塞其他进程。

D、对于聚合查询，可以用HAVING子句进一步限定返回的行。

2、 尽量少做重复的工作

这一点和上一点的目的是一样的，就是尽量减少无效工作，但是这一点的侧重点在客户端程序，需要注意的如下：

A、 控制同一语句的多次执行，特别是一些基础数据的多次执行是很多程序员很少注意的。

B、 减少多次的数据转换，也许需要数据转换是设计的问题，但是减少次数是程序员可以做到的。

C、 杜绝不必要的子查询和连接表，子查询在执行计划一般解释成外连接，

多余的连接表带来额外的开销。

D、 合并对同一表同一条件的多次UPDATE，比如

```
1.UPDATE EMPLOYEE SET FNAME='HAIWER' WHERE EMP_ID=' VPA30890F'
2.3.UPDATE EMPLOYEE SET LNAME='YANG' WHERE EMP_ID=' VPA30890F'
4.5.这两个语句应该合并成以下一个语句

1.UPDATE EMPLOYEE SET FNAME='HAIWER',LNAME='YANG'
2.WHERE EMP_ID=' VPA30890F'
```

E、 UPDATE操作不要拆成DELETE操作+INSERT操作的形式，虽然功能相同，但是性能差别是很大的。

F、 不要写一些没有意义的查询，比如

```
SELECT * FROM EMPLOYEE WHERE 1=2
```

3、 注意事务和锁

事务是数据库应用中和重要的工具，它有原子性、一致性、隔离性、持久性这四个属性，很多操作我们都需要利用事务来保证数据的正确性。在使用事务中我们需要做到尽量避免死锁、尽量减少阻塞。具体以下方面需要特别注意：

A、事务操作过程要尽量小，能拆分的事务要拆分开来。

B、 事务操作过程不应该有交互，因为交互等待的时候，事务并未结束，可能锁定了很多资源。

C、 事务操作过程要按同一顺序访问对象。

D、提高事务中每个语句的效率，利用索引和其他方法提高每个语句的效率可以有效地减少整个事务的执行时间。

E、尽量不要指定锁类型和索引，SQL SERVER允许我们自己指定语句使用的锁类型和索引，但是一般情况下，SQL SERVER优化器选择的锁类型和索引是在当前数据量和查询条件下是最优的，我们指定的可能只是在目前情况下更有，但是数据量和数据分布在将来是会变化的。

F、 查询时可以用较低的隔离级别，特别是报表查询的时候，可以选择最低的隔离级别（未提交读）。

4、 注意临时表和表变量的用法

在复杂系统中，临时表和表变量很难避免，关于临时表和表变量的用法，需要注意：

A、如果语句很复杂，连接太多，可以考虑用临时表和表变量分步完成。

B、 如果需要多次用到一个大表的同一部分数据，考虑用临时表和表变量暂存这部分数据。

C、 如果需要综合多个表的数据，形成一个结果，可以考虑用临时表和表变量分步汇总这多个表的数据。

D、其他情况下，应该控制临时表和表变量的使用。

E、 关于临时表和表变量的选择，很多说法是表变量在内存，速度快，应该首选表变量，但是在实际使用中发现，这个选择主要考虑需要放在临时表的数据量，在数据量较多的情况下，临时表的速度反而更快。

F、 关于临时表产生使用SELECT INTO和CREATE TABLE + INSERT INTO的选择，我们做过测试，一般情况下，SELECT INTO会比CREATE TABLE + INSERT INTO的方法快很多，但是SELECT INTO会锁定TEMPDB的系统表SYSOBJECTS、SYSINDEXES、SYSCOLUMNS，在多用户并发环境下，容易阻塞其他进程，所以我的建议是，在并发系统中，尽量使用CREATE TABLE + INSERT INTO，而大数据量的单个语句使用中，使用SELECT INTO。

G、 注意排序规则，用CREATE TABLE建立的临时表，如果不指定字段的排序规则，会选择TEMPDB的默认排序规则，而不是当前数据库的排序规则。如果当前数据库的排序规则和TEMPDB的排序规则不同，连接的时候就会出现排序规则的冲突错误。一般可以在CREATE TABLE建立临时表时指定字段的排序规则为DATABASE_DEFAULT来避免上述问题。

5、 子查询的用法

子查询是一个 SELECT 查询，它嵌套在 SELECT、INSERT、UPDATE、DELETE 语句或其它子查询中。任何允许使用表达式的地方都可以使用子查询。

子查询可以使我们的编程灵活多样，可以用来实现一些特殊的功能。但是在性能上，往往一个不合适的子查询用法会形成一个性能瓶颈。

如果子查询的条件中使用了其外层的表的字段，这种子查询就叫作相关子查询。相关子查询可以用IN、NOT IN、EXISTS、NOT EXISTS引入。

关于相关子查询，应该注意：

A、NOT IN、NOT EXISTS的相关子查询可以改用LEFT JOIN代替写法。比如：

```
1.SELECT PUB_NAME
2.FROM PUBLISHERS
3.WHERE PUB_ID NOT IN
4.   (SELECT PUB_ID
5.   FROM TITLES
6.   WHERE TYPE = 'BUSINESS')
```

可以改写成：

```
1.SELECT A.PUB_NAME
2.FROM PUBLISHERS A LEFT JOIN TITLES B
3.ON       B.TYPE = 'BUSINESS' AND
4.         A.PUB_ID=B. PUB_ID
5.WHERE B.PUB_ID IS NULL
```

```
1.SELECT TITLE
2.FROM TITLES
3.WHERE NOT EXISTS
4.   (SELECT TITLE_ID
5.   FROM SALES
6.   WHERE TITLE_ID = TITLES.TITLE_ID)
```

可以改写成：

```
1.SELECT TITLE
2.FROM TITLES LEFT JOIN SALES
3.ON SALES.TITLE_ID = TITLES.TITLE_ID
4.WHERE SALES.TITLE_ID IS NULL
```

B、 如果保证子查询没有重复 ， IN、EXISTS的相关子查询可以用INNER JOIN 代替。比如：

```
1.SELECT PUB_NAME
2.FROM PUBLISHERS
3.WHERE PUB_ID IN
4.   (SELECT PUB_ID
5.   FROM TITLES
6.   WHERE TYPE = 'BUSINESS')
```

可以改写成：

```
1.SELECT DISTINCT A.PUB_NAME
2.FROM PUBLISHERS A INNER JOIN TITLES B
3.ON       B.TYPE = 'BUSINESS' AND
4.         A.PUB_ID=B. PUB_ID
```

C、 IN的相关子查询用EXISTS代替，比如

```
1.SELECT PUB_NAME
2.FROM PUBLISHERS
3.WHERE PUB_ID IN
4.   (SELECT PUB_ID
5.   FROM TITLES
6.   WHERE TYPE = 'BUSINESS')
```

可以用下面语句代替：

```
1.SELECT PUB_NAME
2.FROM PUBLISHERS
3.WHERE EXISTS
4.   (SELECT 15.   FROM TITLES
6.   WHERE TYPE = 'BUSINESS' AND
7.   PUB_ID= PUBLISHERS.PUB_ID)
```

D、不要用COUNT(*)的子查询判断是否存在记录，最好用LEFT JOIN或者EXISTS，比如有人写这样的语句：

```
1.SELECT JOB_DESC FROM JOBS
2.WHERE (SELECT COUNT(*) FROM EMPLOYEE WHERE JOB_ID=JOBS.JOB_ID)=0
```

应该改成：

```
1.SELECT JOBS.JOB_DESC FROM JOBS LEFT JOIN EMPLOYEE
2.ON EMPLOYEE.JOB_ID=JOBS.JOB_ID
3.WHERE EMPLOYEE.EMP_ID IS NULL
```

```
1.SELECT JOB_DESC FROM JOBS
2.WHERE (SELECT COUNT(*) FROM EMPLOYEE WHERE JOB_ID=JOBS.JOB_ID)<>0
```

应该改成：

```
1.SELECT JOB_DESC FROM JOBS
2.WHERE EXISTS (SELECT 1 FROM EMPLOYEE WHERE JOB_ID=JOBS.JOB_ID)
```

6、 慎用游标

数据库一般的操作是集合操作，也就是对由WHERE子句和选择列确定的结果集作集合操作，

游标是提供的一个非集合操作的途径。一般情况下，游标实现的功能往往相当于客户端的一个循环实现的功能，所以，大部分情况下，我们把游标功能搬到客户端。

游标是把结果集放在服务器内存，并通过循环一条一条处理记录，对数据库资源（特别是内存和锁资源）的消耗是非常大的，所以，我们应该只有在没有其他方法的情况下才使用游标。

另外，我们可以用SQL SERVER的一些特性来代替游标，达到提高速度的目的。

A、字符串连接的例子

这是论坛经常有的例子，就是把一个表符合条件的记录的某个字符串字段连接成一个变量。比如需要把JOB_ID=10的EMPLOYEE的FNAME连接在一起，用逗号连接，可能最容易想到的是用游标：

```
1.    DECLARE @NAME VARCHAR(20)
2.    DECLARE @NAME VARCHAR(1000)
3.    DECLARE NAME_CURSOR CURSOR FOR
4.    SELECT FNAME FROM EMPLOYEE WHERE JOB_ID=10 ORDER BY EMP_ID
5.    OPEN NAME_CURSOR
6.    FETCH NEXT FROM RNAME_CURSOR INTO @NAME
7.    WHILE @@FETCH_STATUS = 08.    BEGIN
9.      SET @NAMES = ISNULL(@NAMES+',','')+@NAME
10.      FETCH NEXT FROM NAME_CURSOR  INTO @NAME
11.    END
12.    CLOSE NAME_CURSOR
13.    DEALLOCATE NAME_CURSOR
```

可以如下修改，功能相同：

```
1.   DECLARE @NAME VARCHAR(1000)
2.   SELECT @NAMES = ISNULL(@NAMES+',','')+FNAME
3.      FROM EMPLOYEE WHERE JOB_ID=10 ORDER BY EMP_ID
```

B、 用CASE WHEN 实现转换的例子

很多使用游标的原因是因为有些处理需要根据记录的各种情况需要作不同的处理，实际上这种情况，我们可以用CASE WHEN语句进行必要的判断处理，而且CASE WHEN是可以嵌套的。比如：

表结构：

```
1.CREATE TABLE 料件表(
2.料号          VARCHAR(30),
```

```
3.名称          VARCHAR(100),
4.主单位        VARCHAR(20),
5.单位1         VARCHAR(20),
6.单位1参数     NUMERIC(18,4),
7.单位2         VARCHAR(20),
8.单位2参数     NUMERIC(18,4)
9.)
10.11.GO
12.13.CREATE TABLE 入库表(
14.时间             DATETIME,
15.料号             VARCHAR(30),
16.单位             INT,
17.入库数量         NUMERIC(18,4),
18.损坏数量         NUMERIC(18,4)
19.)
20.21.GO
```

其中，单位字段可以是0，1，2，分别代表主单位、单位1、单位2，很多计算需要统一单位，统一单位可以用游标实现：

```
1.DECLARE @料号     VARCHAR(30),
2.        @单位   INT,
3.        @参数     NUMERIC(18,4),
4.5.DECLARE CUR CURSOR FOR
6.        SELECT 料号,单位 FROM 入库表 WHERE 单位 <>07.OPEN CUR
8.FETCH NEXT FROM CUR INTO @料号,@单位
9.WHILE @@FETCH_STATUS<>-110.BEGIN
11.  IF @单位=112.  BEGIN
13.    SET @参数=(SELECT 单位1参数 FROM 料件表 WHERE 料号 =@料号)
14.    UPDATE 入库表 SET 数量=数量*@参数,损坏数量=损坏数量*@参数,单位=1
WHERE CURRENT OF CUR
15.  END
16.  IF @单位=217.  BEGIN
18.    SET @参数=(SELECT 单位1参数 FROM 料件表 WHERE 料号 =@料号)
19.    UPDATE 入库表 SET 数量=数量*@参数,损坏数量=损坏数量*@参数,单位=1
WHERE CURRENT OF CUR
20.  END
21.  FETCH NEXT FROM CUR INTO @料号,@单位
22.END
23.CLOSE CUR
24.DEALLOCATE CUR
```

可以改写成：

```
1.UPDATE A SET
2.数量=CASE A.单位 WHEN 1 THEN      A.数量*B. 单位1参数
3.                 WHEN 2 THEN         A.数量*B. 单位2参数
4.                 ELSE A.数量
5.END,
6.损坏数量= CASE A.单位 WHEN 1 THEN    A. 损坏数量*B. 单位1参数
7.                 WHEN 2 THEN         A. 损坏数量*B. 单位2参数
8.                 ELSE A. 损坏数量
9.END,
10.单位=1
11.FROM入库表 A, 料件表 B
12.WHERE    A.单位<>1      AND
13.         A.料号=B.料号
```

C、 变量参与的UPDATE语句的例子

SQL ERVER的语句比较灵活，变量参与的UPDATE语句可以实现一些游标一样的功能，比如：

在

```
1.SELECT A,B,C,CAST(NULL AS INT) AS 序号
2.INTO #T
3.FROM 表
4.ORDER BY A ,NEWID()
```

产生临时表后，已经按照A字段排序，但是在A相同的情况下是乱序的，这时如果需要更改序号字段为按照A字段分组的记录序号，就只有游标和变量参与的UPDATE语句可以实现了，这个变量参与的UPDATE语句如下：

```
1.DECLARE @A INT
2.DECLARE @序号 INT
3.UPDATE #T SET
4.   @序号=CASE WHEN A=@A THEN @序号+1 ELSE 1 END,
5.   @A=A,
6.   序号=@序号
```

D、如果必须使用游标，注意选择游标的类型，如果只是循环取数据，那就应该用只进游标（选项FAST_FORWARD），一般只需要静态游标（选项STATIC）。

E、 注意动态游标的不确定性，动态游标查询的记录集数据如果被修改，会自动刷新游标，这样使得动态游标有了不确定性，因为在多用户环境下，如果其他进程或者本身更改了纪录，就可能刷新游标的记录集。

7、 尽量使用索引

建立索引后，并不是每个查询都会使用索引，在使用索引的情况下，索引的使用效率也会有很大的差别。只要我们在查询语句中没有强制指定索引，索引的选择和使用方法是SQLSERVER的优化器自动作的选择，而它选择的根据是查询语句的条件以及相关表的统计信息，这就要求我们在写SQL语句的时候尽量使得优化器可以使用索引。

为了使得优化器能高效使用索引，写语句的时候应该注意：

A、不要对索引字段进行运算，而要想办法做变换，比如

```
SELECT ID FROM T WHERE NUM/2=100
```

应改为：

```
SELECT ID FROM T WHERE NUM=100*2
```

```
SELECT ID FROM T WHERE NUM/2=NUM1
```

如果NUM有索引应改为：

```
SELECT ID FROM T WHERE NUM=NUM1*2
```

如果NUM1有索引则不应该改。

发现过这样的语句：

```
1.SELECT 年,月,金额 FROM 结余表
2.WHERE 100*年+月=2007*100+10
```

应该改为：

```
1.SELECT 年,月,金额 FROM 结余表
2.WHERE 年=2007 AND
3.      月=10
```

B、 不要对索引字段进行格式转换

日期字段的例子：

```
WHERE CONVERT(VARCHAR(10), 日期字段,120)=’2008-08-15’
```

应该改为

```
WHERE日期字段〉=’2008-08-15’          AND   日期字段<’2008-08-16’
```

ISNULL转换的例子：

```
WHERE ISNULL(字段,’ ’ )<>’ ’ 应改为:WHERE字段<>’ ’
WHERE ISNULL(字段,’ ’ )=’ ’ 不应修改
WHERE ISNULL(字段,’ F’ ) =’ T’ 应改为: WHERE字段=’ T’
WHERE ISNULL(字段,’ F’ )<>’ T’ 不应修改
```

C、 不要对索引字段使用函数

```
WHERE LEFT(NAME, 3)='ABC' 或者WHERE SUBSTRING(NAME,1, 3)='ABC'
```

应改为:

```
WHERE NAME LIKE 'ABC%'
```

日期查询的例子：

```
WHERE DATEDIFF(DAY, 日期,'2005-11-30')=0应改为:WHERE 日
期 >='2005-11-30' AND 日期 <'2005-12-1 ‘
WHERE DATEDIFF(DAY, 日期,'2005-11-30')>0应改为:WHERE 日期 <'2005-11-30
‘
WHERE DATEDIFF(DAY, 日期,'2005-11-30')>=0应改为:WHERE 日期
<'2005-12-01 ‘
WHERE DATEDIFF(DAY, 日期,'2005-11-30')<0应改为:WHERE 日
期>='2005-12-01 ‘
WHERE DATEDIFF(DAY, 日期,'2005-11-30')<=0应改为:WHERE 日
期>='2005-11-30 ‘
```

D、不要对索引字段进行多字段连接

比如：

```
WHERE FAME+ ’.’+LNAME=‘HAIWEI.YANG’
```

应改为：

```
WHERE FNAME=‘HAIWEI’ AND LNAME=‘YANG’
```

8、 注意连接条件的写法

多表连接的连接条件对索引的选择有着重要的意义，所以我们在写连接条件条件的时候需要特别的注意。

A、多表连接的时候，连接条件必须写全，宁可重复，不要缺漏。

B、 连接条件尽量使用聚集索引

C、 注意ON部分条件和WHERE部分条件的区别

9、 其他需要注意的地方

经验表明，问题发现的越早解决的成本越低，很多性能问题可以在编码阶段就发现，为了提早发现性能问题，需要注意：

A、程序员注意、关心各表的数据量。

B、 编码过程和单元测试过程尽量用数据量较大的数据库测试，最好能用实际数据测试。

C、 每个SQL语句尽量简单

D、不要频繁更新有触发器的表的数据

E、 注意数据库函数的限制以及其性能

10、 学会分辨SQL语句的优劣

自己分辨SQL语句的优劣非常重要，只有自己能分辨优劣才能写出高效的语句。

数据库项目文档

A、 查看SQL语句的执行计划，可以在查询分析其使用CTRL+L图形化的显示执行计划，一般应该注意百分比最大的几个图形的属性，把鼠标移动到其上面会显示这个图形的属性，需要注意预计成本的数据，也要注意其标题，一般都是CLUSTERED INDEX SEEK 、INDEX SEEK 、CLUSTERED INDEX SCAN 、INDEX SCAN 、TABLE SCAN等，其中出现SCAN说明语句有优化的余地。也可以用语句

SET SHOWPLAN_ALL ON

要执行的语句

SET SHOWPLAN_ALL OFF

查看执行计划的文本详细信息。

B、 用事件探查器跟踪系统的运行，可疑跟踪到执行的语句，以及所用的时间，CPU用量以及I/O数据，从而分析语句的效率。

C、 可以用WINDOWS的系统性能检测器，关注CPU、I/O参数

四、 测试、试运行、维护阶段

测试的主要任务是发现并修改系统的问题，其中性能问题也是一个重要的方面。重点应该放在发现有性能问题的地方，并进行必要的优化。主要进行语句优化、索引优化等。

试运行和维护阶段是在实际的环境下运行系统，发现的问题范围更广，可能涉及操作系统、网络以及多用户并发环境出现的问题，其优化也扩展到操作系统、网络以及数据库物理存储的优化。

这个阶段的优花方法在这里不再展开，只说明下索引维护的方法：

A、 可以用DBCC DBREINDEX语句或者SQL SERVER维护计划设定定时进行索引重建，索引重建的目的是提高索引的效能。

B、 可以用语句UPDATE STATISTICS或者SQL SERVER维护计划设定定时进行索引统计信息的更新，其目的是使得统计信息更能反映实际情况，从而使得优化器选择更合适的索引。

C、 可以用DBCC CHECKDB或者DBCC CHECKTABLE语句检查数据库表和索引是否有问题，这两个语句也能修复一般的问题。

D、

五、 网上资料中一些说法的个人不同意见

1、“应尽量避免在 WHERE 子句中对字段进行 NULL 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

```
SELECT ID FROM T WHERE NUM IS NULL
```

可以在NUM上设置默认值0，确保表中NUM列没有NULL值，然后这样查询：

```
SELECT ID FROM T WHERE NUM=0”
```

个人意见：经过测试，IS NULL也是可以用INDEX SEEK查找的，0和NULL是不同概念的，以上说法的两个查询的意义和记录数是不同的。

2、“应尽量避免在 WHERE 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。”

个人意见：经过测试，<>也是可以用INDEX SEEK查找的。

3、“应尽量避免在 WHERE 子句中使用 OR 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：

```
SELECT ID FROM T WHERE NUM=10 OR NUM=20
```

可以这样查询：

```
SELECT ID FROM T WHERE NUM=10

UNION ALL

SELECT ID FROM T WHERE NUM=20”
```

个人意见：主要对全表扫描的说法不赞同。

4、“IN 和 NOT IN 也要慎用，否则会导致全表扫描，如：

```
SELECT ID FROM T WHERE NUM IN(1,2,3)
```

对于连续的数值，能用 BETWEEN 就不要用 IN 了：

SELECT ID FROM T WHERE NUM BETWEEN 1 AND 3”

个人意见：主要对全表扫描的说法不赞同。

5、 “如果在 WHERE 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量,但优化程序不能将访问计划的选择推迟到运行时;它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

SELECT ID FROM T WHERE NUM=@NUM

可以改为强制查询使用索引：

SELECT ID FROM T WITH(INDEX(索引名)) WHERE NUM=@NUM”

个人意见：关于局部变量的解释比较奇怪，使用参数如果会影响性能，那存储过程就该校除了，我坚持我上面对于强制索引的看法。

6、 “尽可能的使用 VARCHAR/NVARCHAR 代替 CHAR/NCHAR ，因为首先变长字段存储空间小,可以节省存储空间,其次对于查询来说,在一个相对较小的字段内搜索效率显然要高些。”

个人意见：“在一个相对较小的字段内搜索效率显然要高些”显然是对的，但是字段的长短似乎不是由变不变长决定，而是业务本身决定。在SQLSERVER6.5或者之前版本，不定长字符串字段的比较速度比定长的字符串字段的比较速度慢很多，所以对于那些版本，我们都是推荐使用定长字段存储一些关键字段。而在2000版本，修改了不定长字符串字段的比较方法，与定长字段的比较速度差别不大了，这样为了方便，我们大量使用不定长字段。

7、 关于连接表的顺序或者条件的顺序的说法，经过测试，在SQL SERVER，这些顺序都是不影响性能的，这些说法可能是对ORACLE有效。

SqlServer 调优综述

公司叫我写sql server性能优化，监控的文档，我这里抄一点，那里抄一点，大家看看写得怎么样！

SQL SERVER 性能调整和优化
一. 应用调整

1． 优化SQL 语句。（见如何正确书写Transact_SQL语句.doc）

2． 在可能时尽量不要在应用中使用数据库游标，游标是非常有用的工具，但是比使用常规的、面向集的SQL 语句需要更大的开销。

3． 避免在同一个系统上混杂联机事务处理（OLTP）和决策支持查询。建议有一个或更多个系统用于OLTP,并有使用脱机数据库的其它系统用于决策支持查询。不要将决策支持和联机事务处理查询混杂到同一个数据库中。

4． 使用存储过程有利于提高系统性能。

二． SQL Server调整

1． 调整处理器

使用Windows NT性能监视器监控系统，以确定是否在某些特定时间，此时处理负担比其他时间要重，这样就可以配置系统处理负载最重的时间。一个常用的规则是如果处理器的使用持续保持在80%或80%以上，或者它经常会到达这个比率，就可能具有一个CPU瓶颈了。这时应更换处理器，或添加处理器，或添加服务器。

2． 调整磁盘驱动器

1)．数据文件对于用户具有极其重要，所以一般考虑数据保护和高可用性，将它们放到RAID5中。数据文件可能执行序列I/O、随机I/O或两者都有。应该尽量将序列I/O分离到与随机I/O不同的磁盘上，同时将不同的序列I/O文件分离到它们各自的磁盘上。序列I/O可以比随机I/O执行得更快，因为磁盘盘片上的搜索时间将大大降低。

即应该将数据文件分布在尽可能多的磁盘驱动器上，以允许更多的并行磁盘访问。

2） 日志文件通常执行序列写，应该和数据文件分开在不同的物理硬盘或磁盘阵列上，建议放在RAID10上，这样也可以提高硬盘I/O性能。并及时备份清空日志。

3） Tempdb应该放在单独的物理硬盘或磁盘阵列上,建议放在RAID 0或RAID0+1上，这样它的性能最高,另外不要对它设置最大值让它自动增长.

4） 备份最好采用网络备份，备份服务器宜采用RAID 10。备份时间应在服务器空闲时。使用多个备份设备，SQL SERVER根据数据文件和备份设备的数量创建许多线程，并行化提高了备份与恢复的性能。

5）创建数据库时创建多个用户定义文件组。为优化系统的磁盘I/O性能，应该将数据分布到尽可能多的磁盘上，以获得更高的并行数据访问(读与写)。文件组的使用可以帮助你达到更均匀地分布I/O。同时建议把用户数据库与系统数据库分开放在不同的文件组。

3． 调整内存

内存调整包括物理内存调整与SQL Server内存调整。在一个专用的SQL Server服务器上，最好使用最小服务器内存和最大服务器内存参数的缺省值，这将允许SQL Server动态地配置其内存池。如果你的服务器还运行有其它的应用,则不能将全部内存都交给SQLSERVER，如果SQL Sserver使用了过多的内存而使其他的应用缺乏内存,则会导致它们进行页面调度。页面调度是一个成本高的操作，应尽可能地减少页面调度。计算能够给予SQL Server的最大可能的内存量，以总的物理内存数减去Windows NT需要的内存总数，再减去非SQL Server使用需要的内存总数。

4. 调整索引

影响到数据库性能的最大因素就是索引。若对该表的查询频率比较高，则应正确建立索引；但是索引不可太多，太多则执行UPDATE DELETE INSERT语句需要用于维护这些索引的开销量急剧增加；避免在索引中有太多的索引键；保证每个索引键值有少数行。维护索引也是很重要的，需要用DBCC 命令整理索引和重建索引。

1）如果对一个表的操作大部分是SELECT 语句，则对此表应建立索引，反之如果有很多INSERT,UPDATE,DELETE则应尽量限制索引数目。

2）索引非常小的表可能反而影响性能。因为这样的表对于SQL SERVER来说，执行一个简单的表扫描更为有效。

3） 尝试建立一些复合索引来进一步提高系统性能。但同时要避免索引中有太多的索引键。因为索引中的数据量越大，对表进行修改时需要更新的数据就越多，因而会引起更多的系统开销。所以应仔细权衡。

4）避免使用大型数据类型的列作为索引键。如：一个整数类型的列就是一个索引键的理想选择，而一个变长字符串类型的列就不是一个理想选择。

5）不宜在一个键值有多个行的列上键索引如：“性别”列。通过命令DBCC SHOW_STATISTICS (table , target)可以察看一个索引的选择性，所返回的密度越低，选择性越大，性能越高。

6）对于具有复合键的聚簇索引，键列在CREATE INDEX语句中列出的顺序也同样重要。当创建这样的一个索引时，数据首先按照第一个键排序，然后根据第二个键排序等等，完全按照你在创建这个索引时所指定的列的顺序。例如，如果你首先在客户表的姓列然后在客户表的名列为客户表创建一个聚簇索引，那么数据首先按姓排序；如果有多个行具有相同的姓，这些行还将按照名被排序。例如，如果有100个具有相同Smith姓的条目，在按姓排序数据后，具有相同Smith姓的行将按照名被排序。因此，如果在WHERE子句中同时包括姓和名，或者在WHERE子句中只使用姓的查询，使用这个索引将更有效率；如果只搜索名，查询效率将会降低，因为对所有匹配的名，都将不得不检查姓。在全部三种情况下，都将使用聚簇索引来满足查询，而且在每种情况下，都会比表上没有索引的执行效率要高。

7）填充因子的设置决定于分页。系统所经历的每秒钟分页的次数显示在SQL Server统计数据Page Splits/sec(分页/秒)中。这个值可以在SQL Server：Access Methods(访问方法)对象下面的性能监视器(Performance Monitor)中找到。如果这个值比较高，就可能需要使用一个较低的填充因子重建索引。对于只读表填充因子为100时最佳。

8)索引视图。SQL Server的一个新特性就是能够对一个视图创建索引，以提高特定的一些查询的性能。主要表现在以下几方面：

a.能够预先计算聚合（如sum,count）并将其存储在索引中，从而最大限度的减少在执行查询期间进行成本很高的计算。

b.能够预先联接表并存储生成的数据集。

c.能够存储联接或聚合的组合。

5. 调整配置参数

可以通过企业管理器或使用T-SQL的sp_configure命令配置大多数参数。以下列举部分常用参数：

1） 相似掩码(affinity mask)参数用以指明在多处理器环境下，SQL Server 可以运行在哪个CPU上。必须关闭并重新启动SQL Server，才能

使这个选项的改变生效。

2） 轻量缓冲池(lightweight pooling)选项用于配置SQL Server使用轻量线程，或纤维。缺省值为0，它禁用纤维。必须关闭并重新启动SQL Server，才能使这个选项的改变生效。

3） 锁(locks)选项用来设置可以获得的最大锁数量，因此限制了SQL Server 用于锁的内存量。缺省设置为0，允许SQL Server基于改变系统的请求，动态地分配和回收锁。必须关闭并重新启动SQL Server，才能使这个选项的改变生效。

4） 最大异步I/O(max async I/O)选项指明每个数据文件上可以进行的未完成的异步I/O的最大数量。缺省值32表明在任何时间，每个文件上可以有32个读和 32个写未完成。必须关闭并重新启动SQL Server，才能使这个选项的改变生效。

5） 最大服务器内存数(max server memory) 为指定SQL Server分配给内存池的最大内存量，以MB为单位。这个选项立即生效，不用关闭和重新启动SQL Server。

6) 最小服务器内存数(min server memory)参数指明了将分配给SQL Server内存池的最小内存数，以MB为单位。缺省值为0，允许SQL Server动态分配和回收内存，这是推荐的设置。这个选项立即生效，不用关闭和重新启动SQL Server。

7）最大工作者线程数(max worker thread)选项指明了SQL Server可以用于处理的最大线程数(或纤维，如果启用了轻量缓冲池)。缺省设置为255。这个选项立即生效，不用关闭和重新启动SQL Server。

8）设置工作区大小(set working set size)参数指明SQL Server分配的、不能为其他的应用使用出页面的内存的数量。必须关闭并重新启动SQL Server，才能使这个选项的改变生效。

9）恢复时间间隔(recovery interval)选项定义了系统在故障时进行恢复花费的最大时间，以分钟为单位。该选项缺省值为0，它指明了SQL Server将替你决定时间间隔—大约一分钟的恢复时间。这个选项立即生效，不用关闭和重新启动SQL Server。

10）用户连接数（user connections ）。SQL Server的用户连接数量不再需要使用配置参数设置—SQL Server可以在需要时动态地分配连接。虽然用户连接在缺省情况下是动态配置的，但也可以使用用户连接数配置参数设置一个最大值。在大多数情况下，你不需要改变这个选项的值。当留给SQL Server动态配置它时，会工作得非常好。必须关闭并重新启动SQL Server，才能使这个选项的改变生效。

三． 硬件调整

1．扩大虚拟内存，并保证有足够可以扩充的空间；把数据库服务器上的不必要服务关闭掉；把数据库服务器和主域服务器分开。在具有一个以上处理器的机器上运行SQL。

2．将硬盘分成NTFS格式，NTFS比FAT32快，并看你的数据文件大小，1G以上你可以采用多数据库文件，这样可以将存取负载分散到多个物理硬盘或磁盘阵列上。

希望能多提宝贵意见！谢谢！

剖析 SQL Server 执行计划

-->Title: 淺議 SQL Server 执行计划

-->Author: wufeng4552
-->Date :2009-10-20 15:08:24
前言:
最近溫習了執行計劃方面的部份知識,為了加深印象與方便初學者,特做了如下整理.

不對地方歡迎提出並指正.

查看执行计划的方式:
(1)菜單方式:
(1.1)显示实际执行计划
(1.2)显示预估的执行计划
以上兩種均位於位于"查询"下拉菜单中,两者的不同之处在于当实际运行一个查询时,当前的服务器上的运算也会被考虑进去。大多数情况下,两种方式产生的执行计划产生的结果是相似的.
(2)命令方式
SET SHOWPLAN_TEXT ON
这条命令被执行后,所有在当前这个查询分析器会话中执行的查询都不会运行,而是会显示一个基于文本的执行计划
注意:执行某条用到临时表的查询时,必须在执行查询先运行 SET STATISTICS PROFILE ON 语句 如:

```
go
if not object_id('tempdb..#t') is null
 drop table #t
Go
Create table #t([日期] Datetime,[姓名] nvarchar(2))
Insert #t
select '2009-10-01',N'张三' union all
```

```
select '2009-10-01',N'李四' union all

select '2009-10-02',N'赵六'

Go
SET STATISTICS PROFILE ON
go
select * from #T
SET STATISTICS PROFILE OFF
```

結果如圖 1

Rows	Exe...	StmtText	StmtId	Nod...	Parent	PhysicalOp	LogicalOp	Argument	DefinedVal...	Estim...	EstimateIO
3	1	select * fro...	1	1	0	NULL	NULL	NULL	NULL	3	NULL
3	1	|--Table Sc...	1	2	1	Table Scan	Table Scan	OBJEC...	[tempdb].[d...	3	0.0032035

圖 1

為了討論方便 下面以 Northwind 庫中表 [Order Details] 為例(我已經將主鍵刪除)

```
use Northwind
go
SET SHOWPLAN_TEXT ON
go
select ProductID,sum(Quantity)Quantity from [Order Details]
group by ProductID order by ProductID
go
SET SHOWPLAN_TEXT OFF
/*
StmtText
|--Sort(ORDER BY:([Northwind].[dbo].[Order
Details].[ProductID] ASC))
     |--Hash Match(Aggregate,
HASH:([Northwind].[dbo].[Order Details].[ProductID])
DEFINE:([Expr1004]=SUM([Northwind].[dbo].[Order
Details].[Quantity])))
```

```
       |--Table Scan(OBJECT:([Northwind].[dbo].[Order
Details]))
*/
use Northwind
go
----建一个聚集索引
CREATE CLUSTERED INDEX INDEX_ProductID on [Order
Details](ProductID)
go
SET SHOWPLAN_TEXT ON
go
select ProductID,sum(Quantity)Quantity from [Order Details]
group by ProductID order by ProductID
go
SET SHOWPLAN_TEXT OFF
/*
StmtText
----------------------------------------
  |--Stream Aggregate(GROUP BY:([Northwind].[dbo].[Order
Details].[ProductID])
DEFINE:([Expr1004]=SUM([Northwind].[dbo].[Order
Details].[Quantity])))
      |--Clustered Index
Scan(OBJECT:([Northwind].[dbo].[Order
Details].[INDEX_ProductID]), ORDERED FORWARD)
(2 個資料列受到影響)
*/
```

如果在执行计划中看到如下所示的任何一项,从性能方面来说,下面所示的每一项都是不理想的。

Index or table scans(索引或者表扫描)：可能意味着需要更好的或者额外的索引。

Bookmark Lookups(书签查找)：考虑修改当前的聚集索引，使用复盖索引，限制 SELECT 语句中的字段数量。

Filter(过滤)：在 WHERE 从句中移除用到的任何函数，不要在 SQL 语句中包含视图，可能需要额外的索引。

Sort(排序)：数据是否真的需要排序？可否使用索引来避免排序？在客户端排序是否会更加有效率？

以上事项避免得越多，查询性能就会越快.

注意:如果有在存储过程中或者其它 T-SQL 批处理代码中用到了临时表，就不能在查询分析器或 Management Studio 使用"显示预估的执行计划"选项来评估查询。必须实际运行这个存储过程或者批处理代码。这是因为使用"显示预估的执行计划"选项来运行一个查询时，它并没有实际被运行，临时表也没有创建。由于临时表没有被创建，参考到临时表的代码就会失败，导致预估的执行计划不能成创建成功。从另一方面来说，如果使用的是表变量而不是临时表，则可以使用"显示预估的执行计划"选项.

```
use Northwind
go
select a.* from [orders] a,[Order Details] b
where a.OrderID=b.OrderID
```

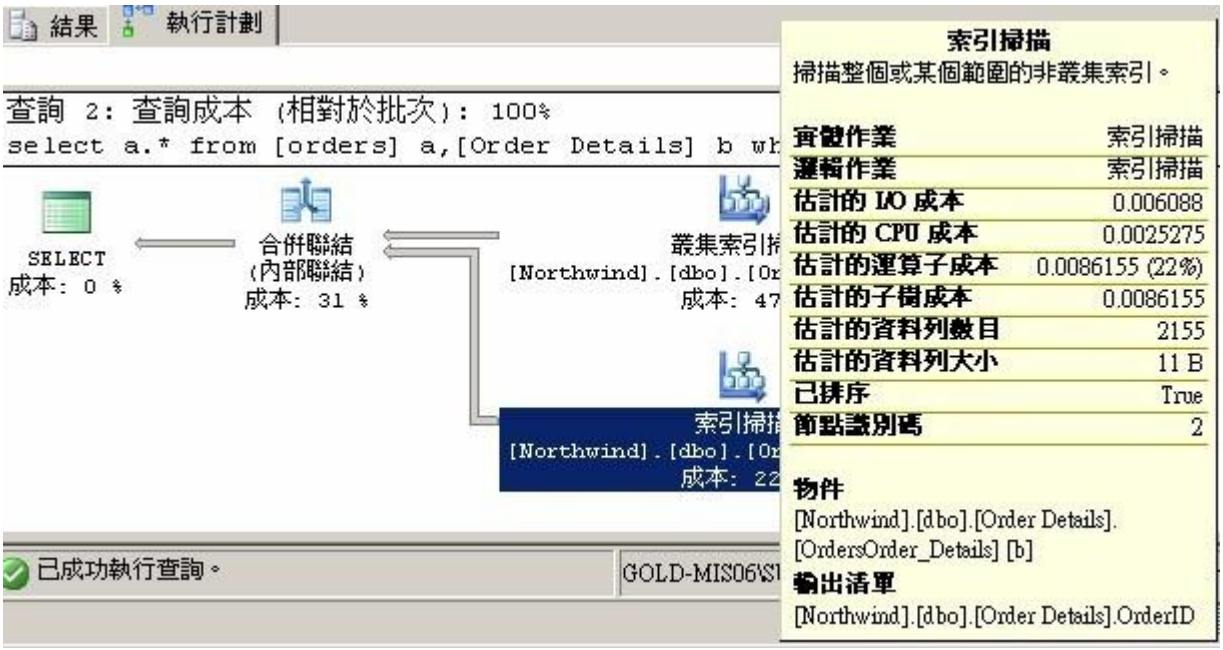


圖 2

查看执行计划时记住如下几点：

(1) 非常复杂的执行计划会被分成多个部分，它们分别列出在屏幕上。每个部分分别代表查询优化器为了得到最终结果而必须执行的单个处理或步骤。执行计划的每个步骤经常会被拆分成一个个更小的子步骤。不幸的是，它们是从右至左显示在屏幕上的。这意味着你必须滚动到图形执行计划的最右边去查看每个步骤是从哪儿开始的

(2) 每个步骤与子步骤间通过箭头连接，藉此显示查询执行的路径。

(3) 最后，查询的所有部分在屏幕顶部的左边汇总到一起,如果将鼠标移动到连接步骤或子步骤的箭头上，就可以看到一个弹出式窗口，上面显示有多少笔记录从一个步骤或子步骤移动到另一个步骤或子步骤(如圖 3) 如果将鼠标移动到任何执行计划任何步骤或者子步骤的上面，就会显示一个弹出式窗口，上面显示该步骤或子步骤的更加详细的信息如圖 2 彈出窗口

圖 3

(4) 图形执行计划上连接每个图标的箭头粗细不同(如圖 3)。箭头的粗细表示每个图标之间移动的数据行数量以及数据行大小移动所需的相对本。箭头越粗，相对成本就越高。可以使用这个指示器来快速测量一个查询。你可能会特别关注粗箭头以了解它如何影响到查询的效能。例如，粗线头应该在图形执行计划的右边，而非左边。如果看到它们在左边，就意味着太多的数据行被返回，这个执行计划也不是最佳的执行计划.

(5) 执行计划的每个部分都被分配了一个成本百分比(如圖 2,3)。它表示这个部分耗用了整个执行计划的多少资源。当对一个执行计划进行分析的时候，应该将精力集中于有着高成本百分比的那些部分。这样就可以在有限的时间里找到可能性最大的问题，从而回报了你在时间上的投资.

(6) 你可能会注意到一个执行计划的某些部分被执行了不止一次。作为执行计划分析的一部分，应该将你的一些时间集中在任何执行了超过一次的那些部分上，看看是否有什么方式减少它们执行的次数。执行的次数越少，查询的速度就越快。

(7) I/O 与 CPU 成本。查询优化器使用这些数字来做出最佳选择。它们可用来参考的一个意义是，较小的 I/O 或 CPU 成本比较大的 I/O 或 CPU 成本使用更少的服务器资源.

```
use Northwind
go
--删除聚集索引
DROP INDEX [Order Details].INDEX_ProductID
--CREATE CLUSTERED INDEX INDEX_ProductID ON [Order
Details](ProductID)
SET STATISTICS IO ON
select * from [Order Details] where ProductID=42
SET STATISTICS IO OFF
```

資料表'Order Details'。掃描計數 1，邏輯讀取 11，實體讀取 0，讀取前讀取 0，LOB 邏輯讀取 0，LOB 實體讀取 0，LOB 讀取前讀取 0。

```
use Northwind
go
--删除聚集索引
--DROP INDEX [Order Details].INDEX_ProductID
--建立聚集索引
```

```
CREATE CLUSTERED INDEX INDEX_ProductID ON [Order
Details](ProductID)
go
SET STATISTICS IO ON
select * from [Order Details] where ProductID=42
SET STATISTICS IO OFF
```

資料表'Order Details'。掃描計數 1，邏輯讀取 2，實體讀取 0，讀取前讀取 0，LOB 邏輯讀取 0，LOB 實體讀取 0，LOB 讀取前讀取 0。

以上可以看出邏輯讀相差很大,由此可以通过 SET STATISTICS IO ON 来查看逻辑读，完成同一功能的不同 SQL 语句，逻辑读越小查询速度越快

(8)将鼠标移到图形执行计划上的表名(以及它的图标)上面，就会弹出一个窗口，从它上面可以看到一些信息。这些信息让你知道是否有用到索引来从表中获取数据，以及它是如何使用的。这些信息包括：

(8.1)Table Scan(表扫描)：如果看到这个信息，就说明数据表上没有聚集索引，或者查询优化器没有使用索引来查找。意即资料表的每一行都被检查到。如果资料表相对较小的话，表扫描可以非常快速，有时甚至快过使用索引。因此，当看到有执行表扫描时，第一件要做的事就是看看数据表有多少数据行。如果不是太多的话，那么表扫描可能提供了最好的总体效能。但如果数据表大的话，表扫描就极可能需要长时间来完成，查询效能就大受影响。在这种情况下，就需要仔细研究，为数据表增加一个适当的索引用于这个查询。假设你发现某查询使用了表扫描，有一个合适的非聚集索引，但它没有用到。这意味着什么呢？为什么这个索引没有用到呢？如果需要获得的数据量相对数据表大小来说非常大，或者数据选择性不高(意味着同一个字段中重复的值很多)，表扫描经常会比索引扫描快。例如，如果一个数据表有 10000 个数据行，查询返回 1000 行，如果这个表没有聚集索引的话，那么表扫描将比使用一个非聚集索引更快。或者如果数据表有 10000 个数据行，且同一个字段(WHERE 条件句有用到这个字段)上有 1000 笔重复的数据，表扫描也会比使用非聚集索引更快。查看图形执行计划上的数据表上的弹出式窗口时，请注意”预估的资料行数(Estimated Row Count)”。这个数字是查询优化器作出的多少个数据行会被返回的最佳推测。如果执行了表扫

描且"预估的数据行数"数值很高的话，就意味着返回的记录数很多，查询优化器认为执行表扫描比使用可用的非聚集索引更快

(8.2)Index Seek(索引查找)：索引查找意味着查询优化器使用了数据表上的非聚集索引来查找数据。性能通常会很快，尤其是当只有少数的数据行被返回时

(8.3)Clustered Index Seek(聚集索引查找)：这指查询优化器使用了数据表上的聚集索引来查找数据，性能很快。实际上，这是 SQL Server 能做的最快的索引查找类型

(8.4)Clustered Index Scan(聚集索引扫描)：聚集索引扫描与表扫描相似，不同的是聚集索引扫描是在一个建有聚集索引的数据表上执行的。和一般的表扫描一样，聚集索引扫描可能表明存在效能问题。一般来说，有两种原因会引此聚集索引扫描的执行。第一个原因，相对于数据表上的整体数据行数目，可能需要获取太多的数据行。查看"预估的数据行数量(Estimated Row Count)"可以对此加以验证。第二个原因，可能是由于 WHERE 条件句中用到的字段选择性不高。在任何情况下，与标准的表扫描不同，聚集索引扫描并不会总是去查找数据表中的所有数据，所以聚集索引扫描一般都会比标准的表扫描要快。通常来说，要将聚集索引扫描改成聚集索引查找，你唯一能做的是重写查询语句，让语句限制性更多，从而返回更少的数据行

(9)绝大多数情况下，查询优化器会对连接进行分析，按最有效率的顺序，使用最有效率的连接类型来对数据表进行连接。但并不总是如此。在图形执行计划中你可以看到代表查询所使用到的各种不同连接类型的图标。此外，每个连接图标都有两个箭头指向它。指向连接图标的上面的箭头代表该连接的外部表，下面的箭头则代表这个连接的内部表。箭头的另一头则指向被连接的数据表名。有时在多表连接的查询中，箭头的另一头指向的并不是一个数据表，而是另一个连接。如果将鼠标移到指向外部连接与内部连接的箭头上，就可以看到一个弹出式窗口，告诉你有多少数据行被发送至这个连接来进行处理。外部表应该总是比内部表含有更少的数据行。如果不是，则说明查询优化器所选择的连接顺序可能不正确

(10) 查看图形执行计划时，你可能会发现某个图标的文字用红色显示，而非通常情况下的黑色。这意味着相关的表的一些统计数据遗失，统计数据是查询优化器生成一个好的执行计划所必须的，遗失的统计数据可以通过右键这个图标，并选择"创建遗失的统计资料"来创建。这时会弹出"创建遗失的统计数据"对话框，通过它可以很容易地创建遗失的统计数据。当可以选择去更新遗失的统计资料时，

应该总是这样做，因为这样极有可能让你正在分析的查询语句从中获得效能上的好处

(11) 有时你会在图形执行计划上看到标识了”Assert”的图标。这意味着查询优化器正在验证查询语句是否有违反引用完整性或者条件约束。如果没有，则没有问题。但如果有的话，查询优化器将无法为该查询建立执行计划，同时会产生一个错误

(12) 你常常会在图形执行计划上看到标识成”书签查找(Bookmark Lookup)”的图标。书签查找相当常见。书签查找的本质是告诉你查询处理器必须从数据表或者聚集索引中来查找它所需要的数据行，而不是从非聚集索引中直接读取。打比方说，如果一个查询语句的 SELECT,JOIN 以及 WHERE 子句中的所有字段，都不存在于那个用来定位符合查询条件的数据行的非聚集索引中，那么查询优化器就不得不做额外的工作在数据表或聚集索引中查找那些满足这个查询语句的字段。另一种引起书签查找的原因是使用了 SELECT *。由于在绝大多情况下它会返回比你实际所需更多的数据，所以应该永不使用 SELECT *.从性能方面来说，书签查找是不理想的。因为它会请求额外的 I/O 开销在字段中查找以返回所需的数据行。如果认为书签查找防碍了查询的性能，那么有四种选择可以用来避免它：可以建立 WHERE 子句会用到的聚集索引，利用索引交集的优势，建立覆盖的非聚集索引，或者(如果是 SQL Server 2000/2005 企业版的话)可以建立索引视图。如果这些都不可能，或者使用它们中的任何一个都会耗用比书签查找更多的资源，那么书签查找就是最佳的选择了。

(13) 有时查询优化器需要在 tempdb 数据库中建立临时工作表。如果是这样的话，就意味着图形执行计划中有标识成 Index Spool, Row Count Spool 或者 Table Spool 的图标。任何时候，使用到工作表一般都会防碍到性能，因为需要额外的 I/O 开销来维护这个工作表。理想情况下应该不要用到工作表。不幸的是并不能总是避免用到工作表。有时当使用工作表比其它选择更有效率时，它的使用实际上会增强性能。不论何种情况，图形执行计划中的工作表都应该引起你的警觉。应该仔细检查这样的查询语句，看看是否有办法重写查询来避免用到工

作表。有可能没有办法。但如果有的话，你就朝提升这个查询的性能方面前进了一步.

(14) 在图形执行计划上看到流聚合(Stream Aggregate)图标就意味着有对一个单一的输入进行了聚合。当使用了 DISTINCT 子句，或者任何聚合函数时，如 AVG, COUNT, MAX, MIN,或者 SUM 等，流聚合操作就相当常见。

(15)查询分析器与 Management Studio 不是唯一的可以生成、显示查询执行计划的工具。SQL Server Profiler 也可以显示执行计划，但格式是文本形式的。使用 SQL Server Profiler 来显示执行计划的一个优势是，它能为实际运行的大量查询产生执行计划。如果使用查询分析器和 Management Studio，则一次只能运行一个

(16)如果在查询中使用了 OPTION FAST 提示，那就必须小心执行计划的结果可能不是你所期望的。这时你所看到的执行计划基于使用了 FAST 提示的结果，而不是整个查询语句的实际执行计划。FAST 提示用来告知果询优化器尽可能快地返回指定行数的数据行，即便这样做会防碍查询的整体性能。使用这个提示的目的在于为使用者快速返回特定行数的记录,由此让他们产生速度非常快速的错觉。。当返回指定行数的数据行后，剩余的数据行按照它们通常的速度返回。因此，如果使用了 FAST 提示,那么生成的执行计划只是基于那些 FAST 返回的数据行，而非查询要返回的所有数据行。如果想看所有数据行的执行计划，那么就必须移除这个 FAST 提示.

SqlServer 性能优化工具

由于现在项目的数据量非常的庞大，日均新增数据几乎达到5位数，数据库的性能就成了整个项目的关键,如何才能做到最优化？不但从数据库逻辑的设计还是物理设计都应该仔细的规划和考虑。

物理上的优化有最笨的方法就是提高数据库的性能，再高个档次就是合理组织数据库文件和Temp数据库存放的位置了，逻辑上优化无非就是有良好的数据库设计数据量大的时候三范式就有一点站不住了，适当的数据冗余也是提高数据库检索效率的有效手段，但这些都需要经验丰富的家伙才能游刃有余，好在SQL2008提供了数据库优化工具使得优化这个工作在一定层面上变得简单了，SQL2000也提供了类似的功能但十分的不可用，2008只是让这个功能可用了而

以，在原理和方法上是一样的，这一点也不得不佩服微软，其一是不完善的功能也敢拿出来用（2000上）其二功能架构的出色设计使得功能有很强的延续性和可扩展性（2005把2000的功能保留并完善）。

优化工具使用很简单

1：使用SQL Server PorFiler（事件查看器）将业务系统中所有操作数据库的步骤都录制下来保存成工作文件。

2：打开sql2008的DataBase Engine Tuning Advisor 将录制下来的作为工作文件然后系统会自动根据你的工作文件来优化数据库。

这种优化就是两种一种是根据工作文件来判断那些表需要建立索引，还有一种就是建立分区，都是从存储结构上来优化数据库的，但它不会告诉你那些SQL语句写的有问题，如果它的只能化最后能到达这个程度就好了。

SqlServer 性能优化工具的使用

```
create table testtable (nkey1 int identity, col2 char(300) default 'abc', ckey1 char(1))
```

接下来，在这个表中填充 10,000 行测试数据。可以为列 nkey1 中所填充的数据创建非聚集索引。可以为列 ckey1 中的数据创建聚集索引，col2 中的数据仅仅是填充内容，将每一行增加 300 字节。

```
declare @counter int

set @counter = 1

while (@counter <= 2000)

begin

insert testtable (ckey1) values ('a')
insert testtable (ckey1) values ('b')
insert testtable (ckey1) values ('c')
insert testtable (ckey1) values ('d')
insert testtable (ckey1) values ('e')
set @counter = @counter + 1
end
```

数据库服务器将进行下面的两个查询：

```
select ckey1,col2 from testtable where ckey1 = 'a'
select nkey1,col2 from testtable where nkey1 = 5000
```

Profiler

SQL Server Profiler 记录数据库服务器中所发生活动的详细信息。可以配置 Profiler 以便用大量的可配置性能信息监视并记录在 SQL Server 中执行查询的一个或多个用户。可在 Profiler 中记录的性能信息有：I/O 统计信息、CPU 统计信息、锁定请求、T-SQL 和 RPC 统计信息、索引和表扫描、警告和引发的错误、数据库对象的创建/除去、连接/断开、存储过程操作、游标操作等等。有关 SQL Profiler 可记录的全部信息，请在 SQL Server Books Online 中搜索字符串“Profiler”。 网管u家bitscn.net

将 Profiler 信息装载到 .trc 文件中以便用于 Index Tuning Wizard 中

Profiler 和 Index Tuning Wizard 是强大的工具组合，以帮助数据库管理员在表中创建适当的索引。Profiler 将查询所消耗的资源记录在 .trc 文件中。.trc 文件可以由 Index Tuning Wizard 读取，Index Tuning Wizard 同时考虑 .trc 信息和数据库表，然后建议应创建什么样的索引。Index Tuning Wizard 可让管理员选择是自动创建数据库的适当索引，调度索引以便在以后自动创建还是产生一个可以在以后查看和执行的 T-SQL 脚本。

以下是分析查询负荷的步骤：

设置 Profiler

从 SQL Server Enterprise Manager 菜单中选择 Tools/SQL Server Profiler 启动 Profiler。

按 CTRL+N 组合键新建 Profiler 跟踪。

键入此跟踪的名称。

选择 Capture to File:复选框，然后选择要将 Profiler 信息输出到其中的 .trc 文件。

单击 OK。

运行工作负荷

启动 Query Analyzer（从 SQL Server Enterprise Manager 菜单中选择 Tools/SQL Server Query Analyzer 或者从开始菜单中选择开始\程序\Microsoft SQL Server 7.0\Query Analyzer）。

连接到 SQL Server 并设定将在其中创建表的当前数据库。

键入或复制以下查询并将它们粘贴到 Query Analyzer 的查询窗口：

```
select ckey1,col2 from testtable where ckey1 = 'a'
```

```
select nkey1,col2 from testtable where nkey1 = 5000
```

按 CTRL+E 执行这两个查询。

停止 Profiler

单击红色的正方形以停止 Profiler 跟踪。

将 .trc 装载到 Index Tuning Wizard

从 Profiler 菜单中选择 Tools\Index Tuning WizardsU 启动 Index Tuning Wizard。单击 Next。

选择要分析的数据库。单击 Next。

保持 I have a saved workload file 选项按钮被选，然后单击 Next。

选择 My workload file 选项按钮，找到用 Profiler 创建的 .trc 文件，然后单击 Next。

在 Select Tables to Tune 对话框中，选择需要进行分析的表，然后单击 Next。

Index Tuning Wizard 将在 Index Recommendations 对话框中指出应创建的索引。单击 Next。

此向导可让您选择是立即创建索引，调度将在以后自动执行的索引创建任务还是创建带创建索引命令的 T-SQL 脚本。选择需要的选项，然后单击 Next。 网管bitscn_com

单击 Finish。

Index Tuning Wizard 为示例数据库和工作负荷生成的 T-SQL。

```
/* Created by:Index Tuning Wizard */
/* Date: 9/7/98 */
/* Time:6:42:00 PM */
/* Server:HENRYLNT2 */
/* Database :test */
/* Workload file :E:\mssql7\Binn\profiler_load.sql */
USE [test]
BEGIN TRANSACTION
CREATE CLUSTERED INDEX [testtable2] ON [dbo].[testtable] ([ckey1])
if (@@error <> 0) rollback transaction
CREATE CLUSTERED INDEX [testtable2] ON [dbo].[testtable] ([ckey1])
if (@@error <> 0) rollback transaction
COMMIT TRANSACTION
```

Index Tuning Wizard 为示例表和数据所建议的索引就是我们预期的索引。ckey1 只有 5 个唯一值，且每一个值都有 2,000 行。假定其中的一个示例查询 (select ckey1, col2 from testtable where ckey1 = 'a') 要求根据 ckey1 中的某个值对表进行检索，那么在 ckey1 列中创建聚集索引是有意义的。第二个查询 (select nkey1, col2 from testtable where nkey1 = 5000) 根据列 nkey1 的值提取一行。Nkey1 唯一，且有 10,000 行，因此在该列创建非聚集的索引是有意义的。 中国网管论坛bbs.bitsCN.com

Profiler/Index Tuning Wizard 组合在涉及许多表和许多查询的实际数据库服务器环境中功能非常强大。当数据库正在进行典型查询时，请使用 Profiler 记录 .trc 文件。然后将 .trc 文件装载到 Index Tuning Wizard，以确定是否创建了正确的索引。根据 Index Tuning Wizard 中的提示自动生成并调度索引创建作业以便在非尖峰时刻运行。定期运行 Profiler/Index Tuning Wizard（比如每周）以查看数据库服务器中所执行的查询是否有较大改动，如果是，则可能需要不同的索引。定期使用 Profiler/Index Tuning Wizard 有助于数据库管理员在查询工作负荷改变以及数据库大小随着时间而增加的同时，保持 SQL Server 以最佳状态运行。

有关详细信息，请在 SQL Server Books Online 中搜索“Index Tuning Wizard”和“Index Tuning Recommendations”。

将 Profiler 信息加载到 SQL Server 表以进行分析

Profiler 提供的另一个选项是将信息记录在 SQL Server 表中。完成后，就可以查询整个表以确定是否有某些查询消耗了过多资源。

将 Profiler 信息记录在 SQL Server 表中

从 SQL Server Enterprise Manager 菜单中选择 Tools/SQL Server Profiler 启动 Profiler。
网管朋友网www_bitscn_net

按 CTRL+N 组合键新建 Profiler 跟踪。
键入跟踪的名称。
单击 Capture to Table:复选框，然后选择要将 Profiler 信息输出到其中的 SQL Server 表。
单击 OK。
结束后，单击红色的正方形停止 Profiler 跟踪。
用 Query Analyzer 分析 Profiler 中记录的信息
在将这些信息记录到 SQL Server 表中后，可以用 Query Analyzer 计算出系统中的哪些查询消耗资源最多。这样，数据库管理员就可以集中时间改进最需要帮助的查询。例如，通常用以下查询分析从 Profiler 记录到 SQL Server 表中的数据。此查询检索数据库服务器中消耗 CPU 资源最多的头 3 项。返回读和写 I/O 信息以及查询的持续时间（用毫秒计）。如果用 Profiler 记录了大量的信息，那么在这个表中创建索引以加快分析查询是有意义的。例如，如果 CPU 即将成为分析这个表的一个重要标准，那么在 CPU 列创建非聚集索引应该是一个不错的主意。

select top 3 TextData,CPU,Reads,Writes,Duration from profiler_out_table order by cpu desc
有关详细信息，请在 SQL Server Books Online 中搜索字符串“Viewing and

Analyzing Traces”、“Troubleshooting SQL Server Profiler”、“Tips for Using SQL Server”、“Common SQL Server Profiler Scenarios”、“Starting SQL Server Profiler”和“Monitoring with SQL Server Profiler”。

网管网www_bitscn_com

Query Analyzer

I/O 统计信息

Query Analyzer 的 Connections Options 对话框 General 选项卡中提供了一个 Show stats I/O 选项。选择此复选框可以获取有关 Query Analyzer 中正在执行的查询所消耗 I/O 量的信息。

例如，当选择 Show stats I/O 选项时，查询“select ckey1, col2 from testtable where ckey1 = 'a'”除返回结果集以外，还返回以下 I/O 信息：

Table 'testtable'.Scan count 1, logical reads 400, physical reads 382, read-ahead reads 400.

同样，当选择 Show stats I/O 选项时，查询“select nkey1, col2 from testtable where nkey1 = 5000”除了返回结果集以外，还返回以下 I/O 信息：

Table 'testtable'.Scan count 1, logical reads 400, physical reads 282, read-ahead reads 400.

使用 I/O 统计信息是一种监视查询调整效果的有效方法。例如，在此示例表中创建 Index Tuning Wizard 在上面所推荐的两个索引，然后再次运行查询。

sqlServer 查询计划和更新计划

c 显示计划(Showplan)是表示由查询优化器生成的文本、图形或 XML 格式的查询计划的术语。他包含了有关 SQL Server 如何处理查询的信息，对查询计划中的每个表，显示计划可以告诉你是否使用了索引，或者是否有必要执行表扫描，以及不同操作的执行顺序。

在本系列随笔的 2.1 中对一个显示计划做了初步的分析。

SQL Server2005 可以生成三种不同格式的显示计划：图形、文本和 XML。

在计划内容方面，SQL Server 可以生成只包含运算符的计划，包含估计成本的计划，以及包含运行时信息的计划。下表列出了生成不同格式计划的命令：

内容	格式		
	文本	XML	图形
运算符	SET SHOWPLAN_TEXT ON	N/A	N/A
运算符和估计成本	SET SHOWPLAN_ALL ON	SET SHOWPLAN_XML ON	在企业管理器中“显示估计的执行计划”
运行时信息	SET STATISTICS PROFILE ON	SET STATISTICS XML ON	在企业管理器中“包含实际的执行计划”

首先看一个简单的查询：

```
SET NOCOUNT ON;

USE Northwind;

GO

SET SHOWPLAN_TEXT ON;

GO

SELECT ProductName, Products.ProductID

FROM dbo.[Order Details]

 JOIN dbo.Products

    ON [Order Details].ProductID = Products.ProductID

WHERE Products.UnitPrice > 100;
```

```
GO

SET SHOWPLAN_TEXT OFF;

GO
```

运行结果：

```
StmtText

------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
----

SELECT ProductName, Products.ProductID

FROM dbo.[Order Details]

 JOIN dbo.Products

    ON [Order Details].ProductID = Products.ProductID

WHERE Products.UnitPrice > 100;



StmtText

------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
----------------

 |--Nested Loops(Inner Join, OUTER
REFERENCES:([Northwind].[dbo].[Products].[ProductID]))

       |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Products].[PK_Products]),
WHERE:([Northwind].[dbo].[Products].[UnitPrice]>($100.0000)))

       |--Index Seek(OBJECT:([Northwind].[dbo].[Order Details].[ProductID]),
SEEK:([Northwind].[dbo].[Order
Details].[ProductID]=[Northwind].[dbo].[Products].[ProductID]) ORDERED FORWARD)
```

输出结果表明：该查询由三个运算符组成：Nested Loops、Clustered Index Scan、Index Seek。

首先看第一句：

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Northwind].[dbo].[Products].[ProductID]))
```

Nested Loops 对两个表进行内部联接，且外部表为 Products 表。

然后是：

```
|--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Products].[PK_Products]),
WHERE:([Northwind].[dbo].[Products].[UnitPrice]>($100.0000)))
```

SQL Server 使用 Clustered Index Scan 访问物理数据，这里扫描聚集索引相当于描述整个表。

最后是：

```
|--Index Seek(OBJECT:([Northwind].[dbo].[Order Details].[ProductID]),
SEEK:([Northwind].[dbo].[Order
Details].[ProductID]=[Northwind].[dbo].[Products].[ProductID]) ORDERED FORWARD)
```

SQL Server 使用 Index Seek 访问索引行，其中的 Object 显示了索引的完整名称。Seek 是查找谓词，这里是要根据外部表的 ProductID 来进行索引查找。

当执行计划时，数据的传递通常是从右到左，从上到下的。缩进多的运算符生成行供缩进少的运算符使用。在这里，Clustered Index Scan 运算符和 Index Seek 运算符是缩进最多的，且 Clustered Index Scan 在 Index Seek 的上面，所以先运算 Clustered Index Scan，然后是 Index Seek，最后是 Nested Loops。

运行示例还可以注意到，当使用了 SET SHOWPLAN_TEXT ON 后，会阻止执行查询。

XML 格式的显示计划

有两种格式的显示计划。一个是 SET SHOWPLAN_XML ON，他将包括估计的执行计划。另一个是 SET STATISTICS XML ON，他包含运行时的信息。

生成 XML 格式的显示计划可以通过以下方法：

1．上面写的这两个指令，其中 SET SHOWPLAN_XML ON 在编译批处理时生成，它为整个批处理生成一个 XML 文档；而 SET STATISTICS XML ON 在运行时产生输出，他为批处理中的每个语句生成单独的 XML 文档。

2．使用企业管理器的“显示图形化显示计划”。

3．使用 SQL Server Profiler。

名为 Showplanxml.xsd 的 XML Schema 描述了包括编译时估计、运行时的 XML 显示计划。在运行时，XML 显示计划提供了一些额外的信息。这个文件在安装完 SQL Server 2005 后，被放置在 Microsoft SQL Server"90"tools"Binn"schemas"sqlserver"2004"07"showplan 目录下。

XML 格式的显示计划的内容最详细，包含了计划大小(CachePlanSize 属性)，和优化该计划是用到的参数值(ParameterList 元素)，而且只有运行时 XML 显示计划才包含并行计划中不同线程所处理的行数(RunTimeCountersPerThread 元素的 ActualRows 属性)，以及执行查询时的实际平行度(DegreeOfParallelism 属性)。

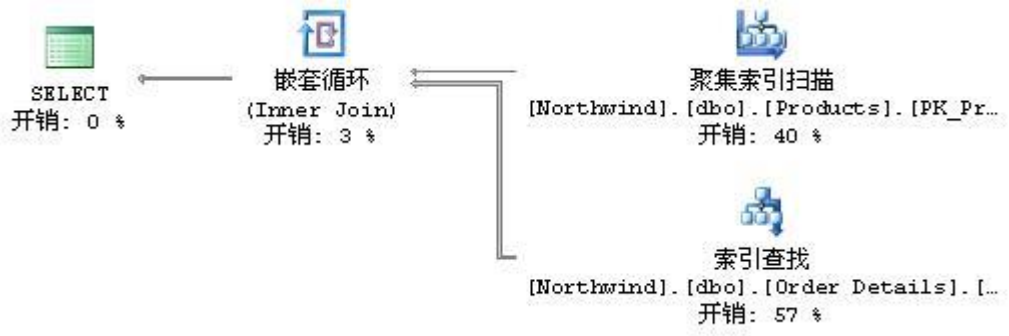
图形化的显示计划

在企业管理器中，有“显示估计的执行计划”和“包括实际的执行计划”两种图形化的显示方法。

显示估计的执行计划：点选后，在结果窗口会立刻显示图形化执行计划。

包含实际的执行计划：点选后，不会立刻显示计划，而是在点击执行后，将实际的计划结果显示出来。

无论使用上面哪种方式图形显示计划，都会显示下面的图形：



但可以想到他们之间存在的区别，当查看运算符的详细信息时：

聚集索引扫描

整体扫描聚集索引或只扫描一定范围。

物理运算	聚集索引扫描
Logical Operation	Clustered Index Scan
估计 I/O 开销	0.003125
估计 CPU 开销	0.0002417
估计运算符开销	0.0033667 (40%)
估计子树大小	0.0033667
估计行数	2
估计行大小	63 字节
已排序	False
节点 ID	1

谓词

[Northwind].[dbo].[Products].[UnitPrice]>($100.0000)

对象

[Northwind].[dbo].[Products].[PK_Products]

输出列表

[Northwind].[dbo].[Products].ProductID, [Northwind].[dbo].[Products].ProductName

聚集索引扫描

整体扫描聚集索引或只扫描一定范围。

物理运算	聚集索引扫描
Logical Operation	Clustered Index Scan
实际行数	2
估计 I/O 开销	0.003125
估计 CPU 开销	0.0002417
估计运算符开销	0.0033667 (40%)
估计子树大小	0.0033667
估计行数	2
估计行大小	63 字节
实际重新绑定次数	0
实际重绕次数	0
已排序	False
节点 ID	1

谓词

[Northwind].[dbo].[Products].[UnitPrice]>($100.0000)

对象

[Northwind].[dbo].[Products].[PK_Products]

输出列表

[Northwind].[dbo].[Products].ProductID, [Northwind].[dbo].[Products].ProductName

因为“包含实际的执行计划”，所以会给出实际运行时的一些信息。

右键->计划另存为，还可以将显示计划保存为 XML 显示计划。扩展名为 sqlplan。

显示计划中的运行时信息

SET STATISTICS XML ON|OFF

XML 显示计划包含两种运行时信息：每个 SQL 语句的信息和每个线程的信息。如果语句有参数，它的计划将包含 ParameterRuntimeValue 属性，表示该语句被执行时每个参数的值。它可能不同于编译该语句时用到的值(ParameterCompiledValue 属性)，但只有优化器在优化并知道该参数值时，该属性才会出现在计划中，且只与传递到存储过程的参数有关。

DegreeOfParallelism 属性表示此次执行的实际平行度(或 DOP，它是单个查询的并发线程数)。它可能与编译时计算的值不同，在编译期间，查询优化器假设为当时的工作负荷为 CPU 的一半。在执行时，DOP 的值会根据执行时开始的工作负荷被调整。如果执行并行计划时 DOP=1，当创建执行上下文时，SQL Server 将从查询计划中移除 Exchange 运算符。MemoryGrant 属性表示以 KB 为单位的用于执行该查询的实际内存。SQL Server 使用这些内存为哈希联接(hash Join)生成哈希表或在内存中执行排序。

RunTimeCountersPerThread 元素包含 5 个属性，每个线程都有相应的值：ActualRebinds、ActualRewinds、ActualRows、ActualEndofScans、ActualExecutions。ActualExecutions 值告诉我们该运算符在每个线程中被初始化的次数。如果运算符是一个扫描运算符，ActualEndofScans 表示扫描到达数据集结尾的次数。所以用 ActualExecutions-ActualEndofScans 就可以得到运算符没有扫描的次数。如：如果 SELECT 中使用 TOP 限定了返回的行数，则输出集合将在扫描到达表的结尾之前被收集。

SET STATISTICS PROFILE ON

这个指令返回的计划与 SET SHOWPLAN_ALL ON 相比有两个区别。他在输出中包含了另外的两列：Rows 和 Executes。

Rows ：是所有线程的 RunTimeCountersPerThread 元素 RowCount 属性的合计，他告诉我们每个运算符实际返回的行数。

Executes：是该元素中 ActualExecutions 属性的合计，他告诉我们 SQL Server 为处理一行或多行而初始化该运算符的次数。

	Rows	Executes	StmtText
1	56	1	SELECT ProductName, Products.ProductID FROM dbo.[Order Details] JOIN dbo....
2	56	1	|--Nested Loops(Inner Join, OUTER REFERENCES:([Northwind].[dbo].[Products].[P...
3	2	1	|--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Products].[PK_Products]), ...
4	56	2	|--Index Seek(OBJECT:([Northwind].[dbo].[Order Details].[ProductID]), SEEK:([N...

当检查某个查询计划时，可以找到查询优化器的估计行数与实际行数之间的最大差异。EstimateRows 列是每次执行所估计的输出行数，而 Rows 是运算符所有执行返回的累积行数。因此我们可以先把 EstimateRows 乘以 EstimateExcutions，在把它与 SET STATISTICS PROFILE 输出的 Roes 列中返回的实际总行数做比较。

用 SQL 跟踪捕获显示计划

使用跟踪来捕获显示计划是非常精确的，这样可以避免在企业管理器中观察到的计划和在应用程序执行起来使用的实际计划之间产生的差异。最常见的，如：用不同的参数调用同一个存储过程、统计信息被自动更新、在编译和运行之间的可用资源(CPU 或内存)发生变化。

使用跟踪进行监视非常消耗资源，监视的事件越多，影响越严重。

下表显示了 9 类性能事件：

跟踪事件类	编译或运行	是否包含运行时信息	是否包含 XML 显示计划	是否为 SQL Server 2000 生成跟踪
Showplan ALL	运行	否	否	是
Showplan All for Query Compile	编译	否	否	否
Showplan Statistics Profile	运行	是	否	是
Showplan Text	运行	否	否	是
Showplan Text(未编码)	运行	否	否	是
Showplan XML	运行	否	是	否

Showplan XML for Query Compile	编译	否	是	否
Showplan XML Statistics Profile	运行	是	是	否
Performance Statistics	编译和运行	是	是	否

如果在开发或调试中，应该使用 Showplan XML Statistics Profile 事件。它生成所有的查询计划和运行时信息。

即使你的服务器很忙，如果设计了有良好的计划重用率的查询，因为编译率较低，也可以使用 Showplan XML For Query Compile 事件。他只在有存储过程或语句被编译或重新编译时才生成跟踪记录，但不包括运行时信息。

通过为各个列设置筛选值可以减少跟踪的大小。在设置跟踪筛选器时，只有 ApplicationName、ClientProcessID、HostName、LogionName、LogionSid、NTDomainName、NTUserName 和 SPID 这些列上应用筛选器会抑制时间的生成。其他筛选器只有在时间被生成并到达客户端后才会应用，所以并不能减少服务器的开销，事实上会造成更多的开销。

另外，相对于在企业管理器中使用显示计划，显示计划跟踪事件进一步扩大了 SQL Server 所能捕获计划的语句集合。如：CREATE、INSERT INTO … EXEC 语句等。

从过程缓存中提取显示计划

上面已经介绍，SQL Server 当生成计划后，会把它保存到过程缓存之中。我们可以用几个动态管理视图和函数、DBCC PROCCACHE、以及目录视图 sys.syscacheobjects 来检查过程缓存。

Sys.dm_exec_query_plan (DMF)以 XML 格式返回位于过程缓存的计划。DMF 要求一个计划句柄作为唯一的参数。计划句柄是一个 VARBINARY(64)类型的查询计划标识符，DMV 为当前过程缓存中每个查询都可以返回该标识符。

看一个示例：

```
SELECT qplan.query_plan AS [Query Plan]

FROM sys.dm_exec_query_stats AS qstats

 CROSS APPLY sys.dm_exec_query_plan(qstats.plan_handle) AS qplan;
```

这个查询为所有缓存的查询计划返回 XML 显示计划。

但是想这样找到某个查询计划非常困难，因为查询文本被包含在 XML 显示计划内部。下面的查询使用 Xquery value 方法从显示计划中提取出序列号(No 列)和查询文本(Statement Text 列)。因为每个批处理都有 sql_handle，所以用 ORDER BY sql_handle 可以按这些语句在批处理中的顺序进行排序显示。

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/showplan' AS sql)

SELECT

 C.value('@StatementId', 'INT') AS [No],

 C.value('(./@StatementText)', 'NVARCHAR(MAX)') AS [Statement Text],

 qplan.query_plan AS [Query Plan]

FROM (SELECT DISTINCT plan_handle FROM sys.dm_exec_query_stats) AS qstats

 CROSS APPLY sys.dm_exec_query_plan(qstats.plan_handle) AS qplan

 CROSS APPLY
query_plan.nodes('/sql:ShowPlanXML/sql:BatchSequence/sql:Batch/
sql:Statements/ descendant::*[attribute::StatementText]') AS T(C)

ORDER BY plan_handle, [No];
```

运行结果：

	No	Statement Text	Query Plan
1	1	SELECT CAST(serverproperty(N'Servername') AS sysna...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
2	1	select value_in_use from sys.configurations where confi...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
3	1	IF (@@microsoftversion / 0x01000000) >= 9 AND ISN...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
4	2	SELECT se.is_admin_endpoint AS N'AdminConnection'...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
5	3	ELSE S...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
6	1	SELECT CAST(serverproperty(N'Servername') AS sysna...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
7	1	SELECT cfg.name AS [Name], cfg.configuration_id AS [...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
8	1	SELECT qplan.query_plan AS [Query Plan] FROM sys....	<ShowPlanXML xmlns="http://schemas.microsoft.com...
9	1	SELECT ProductName, Products.ProductID FROM db...	<ShowPlanXML xmlns="http://schemas.microsoft.com...

更新计划

当优化 INSERT、UPDATE、DELETE 这些数据修改语句时，优化器必须要处理几个特殊的问题。IUD 计划(INSERT、UPDATE、DELETE)包括两个阶段。

第一个阶段：通过生成用于描述数据更改的数据流来确定哪些行将被 IUD。对于 INSERT，数据流包含列值，对于 DELETE，它包含表键，对于 UPDATE，他既包含表键也包含被修改的列的值。

第二个阶段，把数据流中的描述的更改应用到表，通过执行约束验证保证数据完整性，它维护非聚集索引和索引视图，如果存在触发器则引发触发器。

UPDATE 和 DELETE 查询计划通常还包含两个对目标表的引用：第一个引用用于标识受影响的行，第二个引用执行更改的地方。INSERT 计划只包含一个对目标表的引用。

在一些简单的情况是，SQL Server 把 IUD 计划的两个阶段合并在一起。如：把值直接插入表，成为标量插入，或者更新/删除由目标表主键标识的行。

如果 SQL Server 需要执行约束验证，则在第二个阶段会自动包含 Assert 运算符。SQL Server 通过在受影响的行和列上计算一个通常成本较低的标量表达式为 INSERT 和 UPDATE 验证 CHECK 约束。

对包含外键约束的表执行 INSERT 和 UPDATE 会强制验证 CHKECK 约束，而且对包含被引用的表所执行的 INSERT 和 UPDATE 也会强制验证外键约束。为验证约束，即使不是 IUD

操作目标的相关表也被扫描。声明主键将自动地在该列创建唯一的索引，但外键不一样，对被引用的外键列执行 UPDATE 和 DELETE 必须为每个被更新或删除的主键值访问外键表。如果这个约束是一个级联引用完整性约束，那么将会执行更改，否则将验证被删除的键是否存在。因此，要对键值执行 UPDATE 或对主表执行 DELETE，那么应该确保外键上存在索引。

在处理 INSERT 和 DELETE 语句时，除了在聚集索引或堆上执行 IUD 操作，还会维护所有非聚集索引，UPDATE 查询还维护包含被修改列的索引。因为非聚集索引包含聚集索引键和分区键以允许高效地访问数据行，所以更新那些参与聚集索引键或分区键的列成本很高，因为它会修改或重建所有索引。更新分区键还会导致行在分区之间的移动。因此，如果可能的话，应该选择不更新的列作为聚集键或分区键。

总的来说，IUD 语句的性能与包含目标列的索引数量密切相关，因为他们将会被重建或修改。对索引执行单行 INSERT 和 DELETE 操作要求遍历一次索引树。SQL Server 更新索引键和分区键的方法是先执行 DELETE 再执行 INSERT，所以在索引操作上，UPDATE 的成本比 INSERT 和 DELETE 要多一倍。

查询优化器执行 IUD 语句时有两种不同的策略：每行维护和每索引维护。

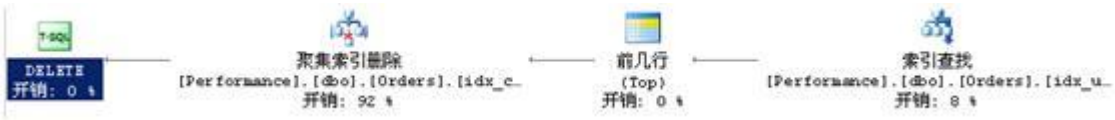
首先使用这个文件准备参考数据：3.rar

然后来看两个查询

查询 1：

```
DELETE FROM dbo.Orders WHERE OrderDate = '2002-01-01'
```

更新计划：



这就是一个每行查询的例子，SQL Server 为该查询所影响的每一行同时维护索引和基表(基表=堆或聚集索引)，并且对所有非聚集索引的更新将与对基表中每一行的更新同时执行。

这个查询计划没有对第 2 个索引执行任何删除操作，因为这些工作在聚集索引删除运算符的计算是同时完成。

查询 2：

```
DELETE FROM dbo.Orders WHERE OrderDate < '2006-01-01'
```

更新计划：

这个查询 2 与查询 1 的更新计划完全不同，因为它执行的是每索引维护。

首先，该计划从聚集索引中删除符合条件的行，同时构建一个临时的假脱机表(spool table)，其中包含必须进行维护的三个非聚集索引的键值。SQL Server 为每个索引读取一次假脱机数据。在读取假脱机数据和从非聚集索引中删除行之间，SQL Server 按被维护索引的顺序排序假脱机数据，以确保对索引页的最佳访问。

Sequence 运算符强制其分支的执行顺序，在这里索引的按从上倒下的顺序进行删除。

每行更新策略在 CPU 方面是高效的，因为同时更新表和所有索引只需很短的代码路径。每索引维护的代码稍微有些复杂，但这样更节省 I/O。如果对键排序后再单独更新非聚集索引，即使同一页中有许多行都被更新，我们也只须访问索引页一次。这也就是为什么 SQL Server 查询优化器认为每行策略需要多次读取被维护索引的同一页才能完成维护，这时它通常选择每索引维护策略。

了解 SQL Server 执行计划

当需要分析某个查询的效能时，最好的方式之一查看这个查询的执行计划。执行计划描述

数据库项目文档

SQL Server查询优化器如何实际运行(或者将会如何运行)一个特定的查询。

查看查询的执行计划有几种不同的方式。它们包括：

SQL Server查询分析器里有一个叫做”显示实际执行计划”的选项(位于”查询”下拉菜单中)。如果打开了这个选项，那么无论何时在查询分析器中运行一个查询，都会得到一个显示在单独窗口的查询执行计划(以图形的格式)。

如果只是想看下执行计划而不想运行查询，那么可以选择”显示预估的执行计划”选项(位于”查询”下拉菜单中)。当选择这个选项后，执行计划会马上显示出来(以图形的格式)。两者的不同之处在于当实际运行一个查询时，当前的服务器上的运算也会被考虑进去。大多数情况下，两种方式产生的执行计划产生的结果是相似的。

当建立一个SQL Server Profiler追踪时，可以收集的一个事件是MISC: Execution Plan.这个信息(以文本的形式)显示查询优化器用来执行查询的计行计划。

可以在查询分析器上执行SET SHOWPLAN_TEXT ON命令。这条命令被执行后，所有在当前这个查询分析器会话中执行的查询都不会运行，而是会显示一个基于文本的执行计划。执行某条用到临时表的查询时，必须在执行查询先运行SET STATISTICS PROFILE ON语句。

上面这些选项中，我更喜欢使用”显示实际执行计划”这个选项。它以图形的方式输出信息，并且考虑到了当前服务器上的那些运算。[7.0, 2000] Updated 8-5-2005

*****

如果在执行计划中看到如下所示的任何一项，就应该将它们视作警告信号并调查它们以找出潜在的性能问题。从性能方面来说，下面所示的每一项都是不理想的。

Index or table scans(索引或者表扫描)：可能意味着需要更好的或者额外的索引。

Bookmark Lookups(书签查找)：考虑修改当前的聚集索引，使用复盖索引，限制SELECT语句中的字段数量。

Filter(过滤)：在WHERE从句中移除用到的任何函数，不要在SQL语句中包含视图，可能需要额外的索引。

Sort(排序)：数据是否真的需要排序？可否使用索引来避免排序？在客户端排序是否会更加有效率？

无一例外地避免这些操作是不可能的，但是避免得越多，查询性能就会越快。
[7.0,2000,2005]

*****

如果有在存储过程中或者其它T-SQL批处理代码中用到了临时表，就不能在查询分析器或Management Studio使用”显示预估的执行计划”选项来评估查询。必须实际运行这个存储过程或者批处理代码。这是因为使用”显示预估的执行计划”选项来运行一个查询时，它并没有实际被运行，临时表也没有创建。由于临时表没有被创建，参考到临时表的代码就会失败，导致预估的执行计划不能成创建成功。

从另一方面来说，如果使用的是表变量而不是临时表，则可以使用”显示预估的执行计划”选项。[7.0,2000,2005] Updated 8-5-2005

*****

数据库项目文档

如果在查询分析器或Management Studio中对一个非常复杂的查询的执行计划进行分析，可能会觉得它的执行计划既难于看懂也难于分析。那么，按照查询的逻辑将它拆分成几个部分，然后分别对这些部分进行分析会容易得多。[7.0,2000,2005] Updated 8-5-2005

*****

图形执行计划并不总是容易读懂和解释。查看执行计划时记住如下几点：

非常复杂的执行计划会被分成多个部分，它们分别列出在屏幕上。每个部分分别代表查询优化器为了得到最终结果而必须执行的单个处理或步骤。

执行计划的每个步骤经常会被拆分成一个个更小的子步骤。不幸的是，它们是从右至左显示在屏幕上的。这意味着你必须滚动到图形执行计划的最右边去查看每个步骤是从哪儿开始的。

每个步骤与子步骤间通过箭头连接，藉此显示查询执行的路径。

最后，查询的所有部分在屏幕顶部的左边汇总到一起。

如果将鼠标移动到任何执行计划任何步骤或者子步骤的上面，就会显示一个弹出式窗口，上面显示该步骤或子步骤的更加详细的信息。

如果将鼠标移动到连接步骤或子步骤的箭头上，就可以看到一个弹出式窗口，上面显示有多少笔记录从一个步骤或子步骤移动到另一个步骤或子步骤。

[7.0, 2000, 2005] Updated 8-5-2005

图形执行计划上连接每个图标的箭头粗细不同。箭头的粗细表示每个图标之间移动的数据行数量以及数据行大小移动所需的相对成本。箭头越粗，相对成本就越高。

可以使用这个指示器来快速测量一个查询。你可能会特别关注粗箭头以了解它如何影响到查询的效能。例如，粗线头应该在图形执行计划的右边，而非左边。如果看到它们在左边，就意味着太多的数据行被返回，这个执行计划也不是最佳的执行计划。[7.0,2000,2005]

*****

执行计划的每个部分都被分配了一个成本百分比。它表示这个部分耗用了整个执行计划的多少资源。当对一个执行计划进行分析的时候，应该将精力集中于有着高成本百分比的那些部分。这样就可以在有限的时间里找到可能性最大的问题，从而回报了你在时间上的投资。[7.0, 2000, 2005]

*****

你可能会注意到一个执行计划的某些部分被执行了不止一次。作为执行计划分析的一部分，应该将你的一些时间集中在任何执行了超过一次的那些部分上，看看是否有什么方式减少它们执行的次数。执行的次数越少，查询的速度就越快。[7.0, 2000, 2005]

*****

在执行计划中你可以看到I/O与CPU成本。它们没有”实际”的意义，例如代表特定资源的使用量。查询优化器使用这些数字来做出最佳选择。它们可用来参考的一个意义是，较小的I/O或CPU成本比较大的I/O或CPU成本使用更少的服务器资源。[7.0, 2000, 2005]

*****

查看SQL Server图形执行计划时，可以查找的非常有用的一个东西就是查询优化器如何为给定的查询使用索引来从表中获取数据。通过查看是否有用到索引，以及索引如何被使用，都有助于判断当前的索引是否使得查询执行得尽可能的快。

将鼠标移到图形执行计划上的表名(以及它的图标)上面，就会弹出一个窗口，从它上面可以看到一些信息。这些信息让你知道是否有用到索引来从表中获取数据，以及它是如何使用的。这些信息包括：

- Table Scan(表扫描)：如果看到这个信息，就说明数据表上没有聚集索引，或者查询优化器没有使用索引来查找。意即资料表的每一行都被检查到。如果资料表相对较小的话，表扫描可以非常快速，有时甚至快过使用索引。

因此，当看到有执行表扫描时，第一件要做的事就是看看数据表有多少数据行。如果不是太多的话，那么表扫描可能提供了最好的总体效能。但如果数据表大的话，表扫描就极可能需要长时间来完成，查询效能就大受影响。在这种情况下，就需要仔细研究，为数据表增加一个适当的索引用于这个查询。

假设你发现某查询使用了表扫描，有一个合适的非聚集索引，但它没有用到。这意味着什么呢？为什么这个索引没有用到呢？如果需要获得的数据量相对数据表大小来说非常大，或者数据选择性不高(意味着同一个字段中重复的值很多)，表扫描经常会比索引扫描快。例如，如果一个数据表有10000个数据行，查询返回1000行，如果这个表没有聚集索引的话，那么表扫描将比使用一个非聚集索引更快。或者如果数据表有10000个数据行，且同一个字段(WHERE条件句有用到这个字段)上有1000笔重复的数据，表扫描也会比使用非聚集索引更快。

查看图形执行计划上的数据表上的弹出式窗口时，请注意”预估的资料行数(Estimated Row Count)”。这个数字是查询优化器作出的多少个数据行会被返回的最佳推测。如果执行了表扫描且”预估的数据行数”数值很高的话，就意味着返回的记录数很多，查询优化器认为执行表扫描比使用可用的非聚集索引更快。

- Index Seek(索引查找)：索引查找意味着查询优化器使用了数据表上的非聚集索引来查找数据。性能通常会很快，尤其是当只有少数的数据行被返回时。

- Clustered Index Seek(聚集索引查找)：这指查询优化器使用了数据表上的聚集索引来查找数据，性能很快。实际上，这是SQL Server能做的最快的索引查找类型。

- Clustered Index Scan(聚集索引扫描)：聚集索引扫描与表扫描相似，不同的是聚集索引扫描是在一个建有聚集索引的数据表上执行的。和一般的表扫描一样，聚集索引扫描可能表明存在效能问题。一般来说，有两种原因会引此聚集索引扫描的执行。第一个原因，相对于数据表上的整体数据行数目，可能需要获取太多的数据行。查看”预估的数据行数量(Estimated Row Count)”可以对此加以验证。第二个原因，可能是由于WHERE条件句中用到的字段选择性不高。在任何情况下，与标准的表扫描不同，聚集索引扫描并不会总是去查找数据表中的所有数据，所以聚集索引扫描一般都会比标准的表扫描要快。通常来说，要将聚集索引扫描改成聚集索引查找，你唯一能做的是重写查询语句，让语句限制性更多，从而返回更少的数据行。

[7.0, 2000, 2005]

绝大多数情况下，查询优化器会对连接进行分析，按最有效率的顺序，使用最有效率的连接类型来对数据表进行连接。但并不总是如此。在图形执行计划中你可以看到代表查询所使用到的各种不同连接类型的图标。此外，每个连接图标都有两个箭头指向它。指向连接图标的上面的箭头代表该连接的外部表，下面的箭头则代表这个连接的内部表。箭头的另一头则指向被连接的数据表名。

有时在多表连接的查询中，箭头的另一头指向的并不是一个数据表，而是另一个连接。如果将鼠标移到指向外部连接与内部连接的箭头上，就可以看到一个弹出式窗口，告诉你有多少数据行被发送至这个连接来进行处理。外部表应该总是比内部表含有更少的数据行。如果不是，则说明查询优化器所选择的连接顺序可能不正确(下面是关于这个话题的更多信息)。

首先，让我们来看看连接类型。SQL Server可以使用三种不同的技术来连接资料表：嵌套循环(nested loop)，散列(hash)，以及合并(merge)。一般来说，嵌套循环是最快的连接类型，但如果不可能使用嵌套循环的话，则会用到散列或者合并作为合适的连接类型。两者都比嵌套循环连接慢。

当连接大表时，则合并连接可能是最佳选项，而非嵌套循环连接。唯一的明确这一点的方式是对两者都进行测试以查看哪一个最有效率。

如果你怀疑某个查询速度慢的原因可能是因为它所使用的连接类型不理想，那么你可以使用连接提示来复盖查询优化器的选择。在使用连接提示之前，你需要花费一些时间去了解一下每种连接类型以及它们的工作方式。这是一个复杂的话题，超出了本文的讨论范围。

查询优化器选择最有效率的连接类型来连接数据表。例如，嵌套循环连接的外部表应该是连接的两个表中较小的那个表。散列连接也是一样，它的外部表应该是较小的那个表。如果你觉得查询优化器选择的连接顺序是错误的，可以使用连接提示来复盖它。

很多情况下，唯一的确认使用连接提示改变连接类型或连接顺序是提升还是降低了效能的方式，就是对它们进行测试，看看发生了什么。[7.0, 2000, 2005]

*****

如果你的SQL Server有多个CPU，并且没有修改SQL Server的默认设置来限制SQL Server使用服务器上所有CPU的能力，那么查询优化器会考虑使用平行处理(parallelism)来执行某些查询。平行处理指在多个CPU上同时运行一个查询的能力。很多情况下，一个运行在多个处理器上的查询比仅运行在单个处理器上的查询要快，但并不总是这样。

查询优化器并不会总是使用平行处理，即使在它能使用的时候。这是因为查询优化器在决定使用平行处理前会考虑到各种不同的因素。例如当前SQL Server上处于活动状态的连接数量，

CPU忙碌程度，是否有足够的内存来运行平行化查询，需要处理的数据行数量，以及这个查询的类型。查询优化器收集到这些真实的数据后，再决定平行处理是不是运行这个查询的最佳选择。你可能会发现，某次一个查询没有用到平行处理，但稍后某次再次运行同样的查询时，却又用到了平行处理。

有时，使用多个处理器所需的花费会大于使用它们能所能节省的资源。尽管查询处理器的确会衡量使用平行查询的正反两面的影响，但它的猜想并不总是正确的。

如果怀疑平行处理防碍了某条查询的性能，你可以使用OPTION (MAXDOP 1)提示来关闭该查询的平行处理。

决定是否使用平行处理的唯一方式是通过这两种方式对查询进行测试，看看发生了什么。[7.0, 2000, 2005]

*****

查看图形执行计划时，你可能会发现某个图标的文字用红色显示，而非通常情况下的黑色。这意味着相关的表的一些统计数据遗失,统计数据是查询优化器生成一个好的执行计划所必须的。

遗失的统计数据可以通过右键这个图标，并选择”创建遗失的统计资料”来创建。这时会弹出”创建遗失的统计数据”对话框，通过它可以很容易地创建遗失的统计数据。

当可以选择去更新遗失的统计资料时，应该总是这样做，因为这样极有可能让你正在分析的查询语句从中获得效能上的好处。[7.0, 2000, 2005]

*****

有时你会在图形执行计划上看到标识了”Assert”的图标。这意味着查询优化器正在验证查询语句是否有违反引用完整性或者条件约束。如果没有，则没有问题。但如果有的话，查询优化器将无法为该查询建立执行计划，同时会产生一个错误。[7.0, 2000, 2005]

*****

你常常会在图形执行计划上看到标识成”书签查找(Bookmark Lookup)”的图标。书签查找相当常见。书签查找的本质是告诉你查询处理器必须从数据表或者聚集索引中来查找它所需要的数据行，而不是从非聚集索引中直接读取。

打比方说，如果一个查询语句的SELECT,JOIN以及WHERE子句中的所有字段，都不存在于那个用来定位符合查询条件的数据行的非聚集索引中,那么查询优化器就不得不做额外的工作在数据表或聚集索引中查找那些满足这个查询语句的字段。

另一种引起书签查找的原因是使用了SELECT *。由于在绝大多情况下它会返回比你实际所需更多的数据，所以应该永不使用SELECT *.

从性能方面来说，书签查找是不理想的。因为它会请求额外的I/O开销在字段中查找以返回所需的数据行。

如果认为书签查找防碍了查询的性能，那么有四种选择可以用来避免它：可以建立WHERE子句会用到的聚集索引，利用索引交集的优势，建立覆盖的非聚集索引，或者(如果是SQL Server

2000/2005企业版的话)可以建立索引视图。如果这些都不可能，或者使用它们中的任何一个都会耗用比书签查找更多的资源，那么书签查找就是最佳的选择了。[7.0, 2000, 2005]

有时查询优化器需要在tempdb数据库中建立临时工作表。如果是这样的话，就意味着图形执行计划中有标识成Index Spool, Row Count Spool或者Table Spool的图标。

任何时候，使用到工作表一般都会防碍到性能，因为需要额外的I/O开销来维护这个工作表。理想情况下应该不要用到工作表。不幸的是并不能总是避免用到工作表。有时当使用工作表比其它选择更有效率时，它的使用实际上会增强性能。

不论何种情况，图形执行计划中的工作表都应该引起你的警觉。应该仔细检查这样的查询语句，看看是否有办法重写查询来避免用到工作表。有可能没有办法。但如果有的话，你就朝提升这个查询的性能方面前进了一步。[7.0, 2000, 2005]。

*****

在图形执行计划上看到流聚合(Stream Aggregate)图标就意味着有对一个单一的输入进行了聚合。当使用了DISTINCT子句，或者任何聚合函数时，如AVG, COUNT, MAX, MIN,或者SUM等，流聚合操作就相当常见。 [7.0, 2000, 2005]

*****

查询分析器与Management Studio不是唯一的可以生成、显示查询执行计划的工具。SQL Server Profiler也可以显示执行计划，但格式是文本形式的。使用SQL Server Profiler来显示执行计划的一个优势是，它能为实际运行的大量查询产生执行计划。如果使用查询分析器和Management Studio，则一次只能运行一个。

使用Profiler捕获、显示执行计划时，必须使用如下的配置生成一个追踪：

捕获事件

- Performance: Execution Plan
- Performance: Show Plan All
- Performance: Show Plan Statistics
- Performance: Show Plan Text

显示的字段

- StartTime
- Duration
- TextData
- CPU
- Reads
- Writes

过滤条件

- Duration。你会想指定最大的查询执行时间，例如5秒钟，由此避免得到太大量的数据。

当然，你可以在你的追踪中捕获更多的没有例在上面的信息，上面例出的只是一个指南而已。但必须记住不要去捕获太多的数据，否则，追踪的运行会影响服务器的性能。[7.0, 2000, 2005]

*****

如果在查询中使用了OPTION FAST提示，那就必须小心执行计划的结果可能不是你所期望的。这时你所看到的执行计划基于使用了FAST提示的结果，而不是整个查询语句的实际执行计划。

FAST提示用来告知果询优化器尽可能快地返回指定行数的数据行，即便这样做会防碍查询的整体性能。使用这个提示的目的在于为使用者快速返回特定行数的记录，由此让他们产生速度非常快速的错觉。。当返回指定行数的数据行后，剩余的数据行按照它们通常的速度返回。

因此，如果使用了FAST提示，那么生成的执行计划只是基于那些FAST返回的数据行，而非查询要返回的所有数据行。如果想看所有数据行的执行计划，那么就必须移除这个FAST提示。[2000,2005]

9.面试问题详解

ETL 四个过程

抽取，清洗，转换，加载

数据仓库构建方法中，ETL的过程和传统的实现方法有一些不同，主要分为四个阶段，分别

是抽取

（extract）、清洗（clean）、一致性处理（comform）和交付（delivery），简称为ECCD。

1．抽取阶段的主要任务是：

读取源系统的数据模型。

连接并访问源系统的数据。

变化数据捕获。

抽取数据到数据准备区。

2．清洗阶段的主要任务是：

清洗并增补列的属性。

清洗并增补数据结构。

清洗并增补数据规则。

增补复杂的业务规则。

建立元数据库描述数据质量。

将清洗后的数据保存到数据准备区。

3．一致性处理阶段的主要任务是：

一致性处理业务标签，即维度表中的描述属性。

一致性处理业务度量及性能指标，通常是事实表中的事实。

去除重复数据。

国际化处理。

将一致性处理后的数据保存到数据准备区。

4．交付阶段的主要任务是：

加载星型的和经过雪花处理的维度表数据。

产生日期维度。

加载退化维度。

加载子维度。

加载1、2、3型的缓慢变化维度。

处理迟到的维度和迟到的事实。

加载多值维度。

加载有复杂层级结构的维度。

加载文本事实到维度表。

处理事实表的代理键。

加载三个基本类型的事实表数据。

加载和更新聚集。

将处理好的数据加载到数据仓库。

ETL即数据抽取（Extract）、转换（Transform）、装载（Load）的过程。它是构建数据仓库的重要环节。数据仓库是面向主题的、集成的、稳定的且随时间不断变化的数据集合，用以支持经营管理中的决策制定过程。数据仓库系统中有可能存在着大量的噪声数据，引起的主要原因有：滥用缩写词、惯用语、数据输入错误、重复记录、丢失值、拼写变化等。即便是一个设计和规划良好的数据库系统，如果其中存在着大量的噪声数据，那么这个系统也是没有任何意义的，因为“垃圾进，垃圾出”（garbage in, garbage out），系统根本就不可能为决策分析系统提供任何支持。为了清除噪声数据，必须在数据库系统中进行数据清洗。目前有不少数据清洗研究和ETL研究，但是如何在ETL过程中进行有效的数据清洗并使这个过程可视化，此方面研究不多。本文主要从两个方面阐述ETL和数据清洗的实现过程：ETL的处理方式[19]和数据清洗的实现方法。

数据库项目文档

（1）ETL的处理方式

本文所采用的ETL方法是数据库段区域中的ETL处理方式，它不使用外部引擎而是使用数据库作为唯一的控制点。由于源系统SQLserver2000是关系数据库，它的段表也是典型的关系型表。成功地将外部未修改数据载入数据库后，再在数据库内部进行转换。数据库段区域中的ETL处理方式执行的步骤是提取、装载、转换，即通常所说的ELT。[21]这种方式的优点是为抽取出的数据首先提供一个缓冲以便于进行复杂的转换，减轻了ETL进程的复杂度。

（2）ETL过程中实现数据清洗的实现方法

首先，在理解源数据的基础上实现数据表属性一致化。为解决源数据的同义异名和同名异义的问题，可通过元数据管理子系统，在理解源数据的同时，对不同表的属性名根据其含义重新定义其在数据挖掘库中的名字，并以转换规则的形式存放在元数据库中，在数据集成的时候，系统自动根据这些转换规则将源数据中的字段名转换成新定义的字段名，从而实现数据挖掘库中的同名同义。

其次，通过数据缩减，大幅度缩小数据量。由于源数据量很大，处理起来非常耗时，所以可以优先进行数据缩减，以提高后续数据处理分析效率。

最后，通过预先设定数据处理的可视化功能节点，达到可视化的进行数据清洗和数据转换的目的。针对缩减并集成后的数据，通过组合预处理子系统提供各种数据处理功能节点，能够以可视化的方式快速有效完成数据清洗和数据转换过程。

ETL即数据抽取（Extract）、转换（Transform）、装载（Load）的过程。它是构建数据仓库的重要环节。数据仓库是面向主题的、集成的、稳定的且随时间不断变化的数据集合，用以支持经营管理中的决策制定过程。数据仓库系统中有可能存在着大量的噪声数据，引起的主要原因有：滥用缩写词、惯用语、数据输入错误、重复记录、丢失值、拼写变化等。即便是一个设计和规划良好的数据库系统，如果其中存在着大量的噪声数据，那么这个系统也是没有任何意义的，因为“垃圾进，垃圾出”（garbage in，garbage out），系统根本就不可能为决策分析系统提供任何支持。为了清除噪声数据，必须在数据库系统中进行数据清洗。目前有不少数据清洗研究和ETL研究，但是如何在ETL过程中进行有效的数据清洗并使这个过程可视化，此方面研究不多。本文主要从两个方面阐述ETL和数据清洗的实现过程：ETL的处理方式[19]和数据清洗的实现方法。

（1）ETL的处理方式

本文所采用的ETL方法是数据库段区域中的ETL处理方式，它不使用外部引擎而是使用数据库作为唯一的控制点。由于源系统SQLserver2000是关系数据库，它的段表也是典型的关系型表。成功地将外部未修改数据载入数据库后，再在数据库内部进行转换。数据库段区域中的ETL处理方式执行的步骤是提取、装载、转换，即通常所说的ELT。[21]这种方式的优点是为抽取出的数据首先提供一个缓冲以便于进行复杂的转换，减轻了ETL进程的复杂度。

（2）ETL过程中实现数据清洗的实现方法

首先，在理解源数据的基础上实现数据表属性一致化。为解决源数据的同义异名和同名异义的问题，可通过元数据管理子系统，在理解源数据的同时，对不同表的属性名根据其含义重新定义其在数据挖掘库中的名字，并以转换规则的形式存放在元数据库中，在数据集成的时候，系统自动根据这些转换规则将源数据中的字段名转换成新定义的字段名，从而实现数据挖掘库中的同名同义。

其次，通过数据缩减，大幅度缩小数据量。由于源数据量很大，处理起来非常耗时，所以可

以优先进行数据缩减，以提高后续数据处理分析效率。

最后，通过预先设定数据处理的可视化功能节点，达到可视化的进行数据清洗和数据转换的目的。针对缩减并集成后的数据，通过组合预处理子系统提供各种数据处理功能节点，能够以可视化的方式快速有效完成数据清洗和数据转换过程。

逻辑数据映射，及对项目组的作用

逻辑数据映射是用来描述源系统的数据定义，目标数据仓库的模型以及将源系统数据转换数据仓库中需要做的操作和处理。他对ETL项目组起着元数据的作用。

还原数据库的脚本。

```
Restore database 数据库名 from disk='备份文件'
```

SQL 数据库对象。

表，视图，函数，存储过程，索引，触发器，

SQL 索引。

索引是一种特殊的文件，就是数据表中的数据和相应的存储位置的表，可以加快数据减操作数据库但数据修改操作会变慢。

查询当月的天数

```
Select datediff(dd,getdate(),dateadd(mm,1,getdate()))
```

临时表和表变量的区别

1．表变量的操作并不记录在transaction log里，所以对表变量不能用rollback，而临时表则可以．

2．任何有临时表的过程都不能被预编译(pre-compiled)，但是有表变量的过程的execution plan却有可能被预先生成．所以从这个意义来看，表变量的过程要比临时表的过程速度要快．

•表变量缺省放在内存，速度快，所以在触发器，存储过程里如果数据量不大，应该用表变量。

•临时表是利用了硬盘(tempdb数据库），表名变量是占用内存，因此小数据量当然是内存中的

表变量更快。当大数据量时，就不能用表变量了，太耗内存了。大数据量时适合用临时表。
•临时表相对而言表变量主要是多了I/O时间，但少了对内存资源的占用。数据量较大的时候，由于对内存资源的消耗较少，使用临时表比表变量有更好的性能。
•表变量有明确的作用域，在定义表变量的函数、存储过程或批处理结束时，会自动清除表变量
•涉及表变量的事务只在表变量更新期间存在。这样就减少了表变量对锁定和记录资源的需求。

1 临时表可以支持事务级的回滚操作（undo），但不支持前滚操作(redo)，表变量不支持事务级的回滚操作，只是支持语句级的回滚。

2 临时表上的统计信息是健全而可靠的，但是表变量上的统计信息是不可靠的。

3 临时表的编译阀值非常低，但是表变量的编译阀值更低。这个后朋友寻觅经过测试，表变量是不存在编译阀值的。

4 范围不同。临时表为会话级，表变量为变量级，在自己的代码声明区间内有效。

5 表变量只能指定primary key做隐式的索引，但是临时表可以显式创建各种索引。

6 在2000和2005的区别里，表变量和临时表的差异更加显著。

7 表变量不支持修改自身的定义。

8 修改表变量的语句中，不能使用并行的执行计划。

在我的实际编程调优经验中，表变量一般只是应用于超小型表，比如100行以内，不然表变量极易引起重编译和未知的性能问题。

不过具体的行数可以根据实际情况，使用临时表和表变量两种方案对比，哪个更快再来作出最切合实际的决定。

Delete 和 Truncate 的区别

truncate不记录日志,不能从日志中恢复数据

TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。

DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据，并且只在事务日志中记录页的释放。

TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。

对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

TRUNCATE TABLE 不能用于参与了索引视图的表。

相同点:truncate和不带where子句的delete, 以及drop都会删除表内的数据

不同点:

1. truncate和 delete只删除数据不删除表的结构(定义)

drop语句将删除表的结构被依赖的约束(constrain),触发器(trigger),索引(index); 依赖于该表的存储过程/函数将保留,但是变为invalid状态.

2.delete语句是dml,这个操作会放到rollback segement中,事务提交之后才生效;如果有相应的trigger,执行的时候将被触发.

truncate,drop是ddl, 操作立即生效,原数据不放到rollback segment中,不能回滚. 操作不触发trigger.

3.delete语句不影响表所占用的extent, 高水线(high watermark)保持原位置不动

显然drop语句将表所占用的空间全部释放

truncate 语句缺省情况下将空间释放到 minextents个 extent,除非使用reuse storage; truncate会将高水线复位(回到最开始).

4.速度,一般来说: drop> truncate > delete

5.安全性:小心使用drop 和truncate,尤其没有备份的时候.否则哭都来不及

使用上,想删除部分数据行用delete,注意带上where子句. 回滚段要足够大.

想删除表,当然用drop

想保留表而将所有数据删除. 如果和事务无关,用truncate即可. 如果和事务有关,或者想触发trigger,还是用delete.

如果是整理表内部的碎片,可以用truncate跟上reuse stroage,再重新导入/插入数据

事物有哪一种

有三种。第一种是显式声名的事务,这种事务要以BEGIN TRANSACTION为事务的起始标志。第二种是自动提交事

务，这是SQL Server的缺省设置。每一个T－SQL语句在执行完成后会被自动提交。第三种是隐含事务，在这种方式下，

SQL Server会在当前事务被提交或回滚后自动启动一个新的事务，这个新事务直到用户执行COMMIT或ROLLBACK为止,这时系

统又会启动一个新事务。这样就形成了一个连续的事务链。

游标的分类

MS SQL SERVER 支持三种类型的游标：Transact_SQL 游标，API 服务器游标和客户游标。

（1） Transact_SQL 游标

Transact_SQL 游标是由DECLARE CURSOR 语法定义、主要用在Transact_SQL 脚本、存储过程和触发器中。Transact_SQL 游标主要用在服务器上，由从客户端发送给服务器的Transact_SQL 语句或是批处理、存储过程、触发器中的Transact_SQL 进行管理。Transact_SQL 游标不支持提取数据块或多行数据。

（2） API 游标

API 游标支持在OLE DB， ODBC 以及DB_library 中使用游标函数，主要用在服务器上。每一次客户端应用程序调用API 游标函数，MS SQL SEVER 的OLE DB 提供者、ODBC 驱动器或DB_library 的动态链接库（DLL） 都会将这些客户请求传送给服务器以对API游标进行处理。

（3） 客户游标

客户游标主要是当在客户机上缓存结果集时才使用。在客户游标中，有一个缺省的结果集被用来在客户机上缓存整个结果集。客户游标仅支持静态游标而非动态游标。由于服务器游标并不支持所有的Transact-SQL 语句或批处理，所以客户游标常常仅被用作服务器游标的辅助。因为在一般情况下，服务器游标能支持绝大多数的游标操作。

由于API 游标和Transact-SQL 游标使用在服务器端，所以被称为服务器游标，也被称为后台游标，而客户端游标被称为前台游标。在本章中我们主要讲述服务器（后台）游标。