**Over子句**

Over子句用于为行定义一个窗口（window），以便进行特定的运算。可以把行的窗口简单地认为是运算符的一个行的集合。例如聚合函数和排名函数都是可以支持Over子句的运算类型。由于Over为这些函数提供了一个行的接口，所以这些函数也被称之为开窗函数（window function）。

聚合开窗函数使用Over子句提供的窗口作为上下文，对窗口中的一组值进行操作，而不是使用Group By子句提供的上下文。这样就不必对数据进行分组，还能够在同一行中同时返回基础行的列和聚合列。

如果我们想对行进行限制或分区，则可以使用Partition By子句。例如，现在不想返回所有的行，只是想返回当前用户的总价格，则可以指定Sum(val) Over(Partition By custid)

Over子句也支持四种排名函数：Row\_Number(行号) ，Rank(排名)，DENSE\_RANK(密集排名)，以及NTILE.

Row\_Number函数用于为查询的结果集中的各行分配递增的序列号，其逻辑顺序通过Over子句中的Order By语句进行指定。

NTILE函数可以把结果中的行关联到组，并为每一行分配一个所属组的编号。NTILE接收一个表示组数量的输入参数，并要在Over子句中指定逻辑顺序。

**谓词**

T-SQL支持的谓词包括IN，Between和Like等。

IN这个谓词用于检查一个值是否与一组元素中的至少一个相等。

Between这个谓词用于检查一个值是否在指定范围内，包括两个指定的边界值。

Like这个谓词用于检查一个字符串值是否与指定模式匹配。

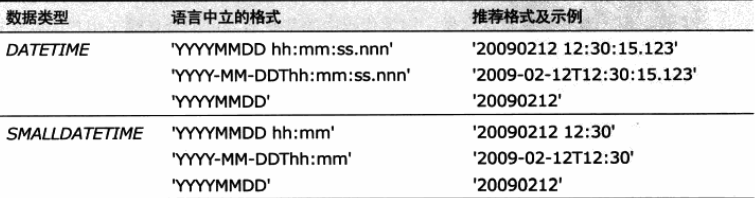
**NULL值**

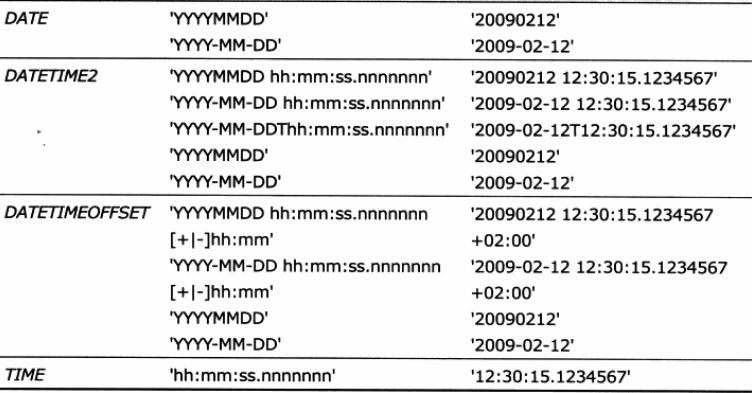
在where子句中，如果结果需要包含NULL，则必须指定：ISNULL(字段)。因为条件查询时会过滤掉UNKNOWN的值。

**日期格式和转换，比较**

**日期格式转换**

字符串转日期格式时，强烈推荐按照语言无关性的方式来编写日期和时间字符串文字。下面列举了每种日期和时间类型的语言无关的字符串格式：





SQL语句如下：

SELECT CAST('2018-01-23T10:23:01' AS DATETIME)

--2018-01-23 10:23:01.000

SELECT CAST('20181001' AS SMALLDATETIME)

--2018-10-01 00:00:00

SELECT CAST('20000901 07:30:00' AS DATETIME2)

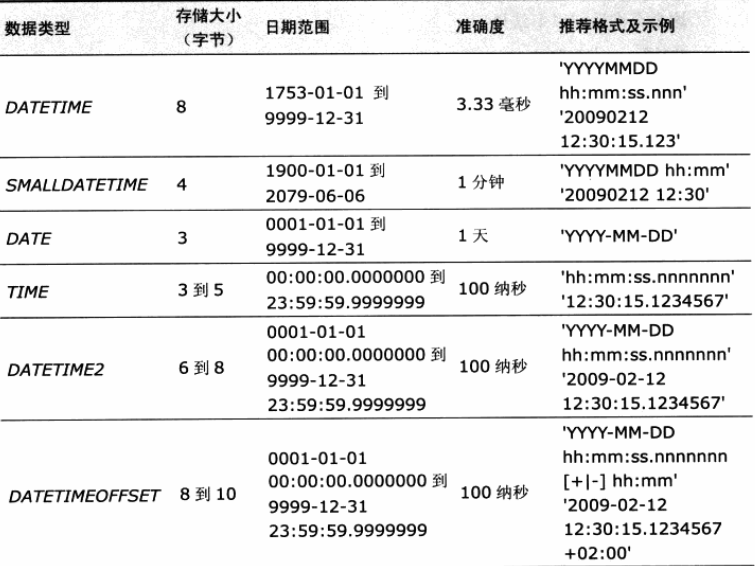
2000-09-01 07:30:00.0000000

SELECT CAST('19:45:00' AS TIME)

--19:45:00.0000000

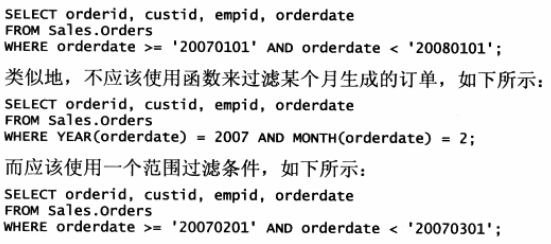
**日期类型**

SQLServer2008引入了新的日期类型：DateTime2，一种比DateTime具有更大的日期范围和更好精度的日期类型；以及DATETIMEOFFSET它具有一个时区的组成部分。



**过滤日期条件**

SQLS如果要过滤日期条件，比较自然的方法是调用Month和Year之类的函数。不过应该注意的一点是：在大多数情况下，当对过滤条件的列做了一些处理之后，就不能以有效的方式使用索引了。所以我们需要对谓词进行调整，以便对过滤条件中的列进行处理。如下所示：



**联接查询**

Join表操作符对两个输入表进行了操作。联结有三种基本类型：交叉连接，内连接，外链接。这三种连接的区别是它们采用的逻辑查询处理步骤各部相同，每种连接都有一套不同的步骤。交叉连接只有一个步骤—笛卡尔积；内连接有两个步骤—笛卡尔积，过滤；外链接有三个步骤—笛卡尔积，过滤，添加外部行。

**交叉连接**

交叉联接是一种最简单的联接，交叉联接只实现一个笛卡尔积即可。这一步就是对两张表进行操作，把它们联接起来，生成两者的笛卡尔积。也就是将一个输入表的每行和另一张表的每行进行匹配。如果一张表有n行，另一张表有m行，将得到m \* n行。

ANSI SQL-92：Cross Join

ANSI SQL-89：在两张表间放置一个逗号

**内连接**

内连接要应用两个逻辑查询处理步骤：它首先像交叉联接一样，对两个输入表进行笛卡尔积运算：然后根据用户指定的谓词对结果进行过滤。和交叉联接一样，内联接也有两种标准语法。

ANSI SQL-92：Inner Join 或者 Join（由于内连接是默认的联接方式，所以也可以只写Join）

ANSI SQL-89：在两张表间放置一个逗号，在查询的where子句中写过滤条件。

强烈推荐使用Inner Join，因为假如你忘记写过滤条件(on)，执行时SQL会报错。而如果使用SQL-89语法，则不会报错。那么这个内连接就变成交叉连接了! 而且实际开发中SQL语句会非常的复杂，写错的概率就很大了。

**组合连接**

有时候过滤条件不止一个，这时候就要用and来连接多个条件，这就叫做组合连接。例如：Table2表中定义了一个外键（col1和col2），引用了Table1表的col1和col2列，现在要写一个根据主外键关系来连接两个表的查询语句。该条件如下：

FROM TABLE1 AS T1

JOIN TABLE2 AS T2

ON T1.col1 = T2.col1 AND T1.col2 = T2.col2

**不等连接**

如果连接条件包含等号运算符，那么这样的连接叫做等值连接（equi join）。如果连接条件包含除等号以外的其他运算符，那么这样的连接叫做不等连接（non-equi join）。

**多表连接**

一个联接表运算符只对两个表进行操作，而一条查询语句可以包含多个联接。通常，当FROM子句中包含多个表运算符时，表运算符在逻辑上是按从左到右的顺序处理的。也就是说，第一个表运算符的结果将作为第二个表运算符的输入，第二个表运算符的结果将作为第三个运算符左边的输入，以此类推。

**外连接**

外连接会应用内连接所应用的两个逻辑处理步骤（笛卡尔积和On过滤），此外还多一个外连接特有的第三步：添加外部行。

在外联接中，耍把一个表标记为“保留的”表，可以在表名之间使用关键字LEFT OUT JOIN、RIGHT OUTER JOIN ,以及 FULL OUTER JOIN,其中 OUTER 失键字是可选的。 LEFT关键字表示左边表的行是保留的关键字表示右边表的行是保留的，而FULL 关键字则表示左右两边表的行都是保留的。外联接的第三个逻辑杏询处理步骤就是要识别保留表中按照ON条件在另一个表找不到与之匹配的那些行，再把这些行添加到联接 的前两个步骤生成的结果表中：对于来自联接的非保留表的那些列，追加的外部行中的 这些列则用NULL作为占位符。

**外连接常见问题**

**1.对外联接中非保留表的列进行过滤**

当检查涉及外联接的代码时，查找逻辑错误时，应该检查的一个地方就是WHERE子句。如果WHERE子句的条件是以<列><运算符><值>的形式引用了联接中的非保留表的列。这通常就是存在错误的一个标志。因为外联接得到的外部行中来自非保留字的列值均为NULL，而NULL <运算符><值>这种格式只会得到UNKNOWN（除非使用IS NULL运算符，显示查找NULL值）。这种结果相当于抵消了外联接的作用。

**2.在多表连接中使用外连接**

在From子句中运行的表运算符并不适用与“同时操作”，表运算符逻辑上是从左到右计算的。对外联接的处理顺序的调整，可能会影响到输出结果，所以不能随便改变它们的顺序。

例如：假设现在要写一条多表联接查询语句，先对两张表进行外联接，再和第三种表进行内连接。如果在内连接ON子句中的条件是对来自外联接非保留表的列和第三张表的列进行比较，那么所有的外部行都会被过滤掉。

记住，对于外部行，其来自非保留表的列值为NULL；把NULL和任何值进行比较，得到的都是UNKNOWN;UNKNOWN都会被ON过滤掉。

那么我们如何避免这种情况呢？我们可以采用调换顺序的方式来解决这个问题，或者采用（）来让内连接先执行，最后在外联接。

**2.随外联接一起使用COUNT聚合函数**

另一个常见的逻辑错误是与外联接一起使用COUNT聚合函数有关。当对外连接的结果进行分组，再使用COUNT(\*)聚合函数时，聚合操作会把内部行和外部行都计算在内，因为它不计算行数，而不管行的内容。但通常是不希望把外部行也纳入计数的考虑范围之内。

为了解决这一个问题，应该用COUNT(<COLUMN>)来代替COUNT(\*)，并从联接的非保留表中选择一个列。这样一来 COUNT就会忽略外部行，因为这些行在计数列上的值为NULL。记住要选择只有在外部行中列值为NULL的列。

**子查询**

SQL支持在查询语句中编写查询，或者嵌套其他查询。最外层查询的结果集会返回给调用者，称为外部查询。

内部查询的结果是供外部查询使用的，也称为子查询。

子查询也可以分为独立子查询（self-contained subquery）和相关子查询（correlated subquery）两类。独立子查询不依赖于它所属的外部查询，而相关子查询则依赖与它所属的外部查询。

**独立子查询**

所有子查询都有所属于的外部查询。独立子查询是独立于其外部查询的子查询。独立子查询调试起来非常方便，因为总可以把子查询代码独立出来单独运行，并确保它能够正确实现默认的功能。

在逻辑上，独立子查询在执行外部查询之前只要先执行一次，接着外部查询再使用子查询的结果继续进行查询。

标量子查询是返回单个值的子查询。

多值子查询是在一个列中返回多个值的子查询。一些谓词如IN可以处理多值子查询。

在许多情况下，我们既可以使用子查询也可以使用表联接来解决问题。在一些情况下，子查询和联接是一样的；而在另一些情况下，对两者的解释是不同的。我的方法是：对于给定的任务，先用直观的形式写出能解决问题的语句，如果对它的性能不满意，调整方法之一就是重构查询。这样的查询重构可以包括用子查询代替联接，也可以是联接代替子查询。

**相关子查询**

相关子查询是指引用了外部查询中出现的表的列的子查询。这就意味着子查询要依赖于外部查询，不能独立地调用它。在逻辑上，子查询会为外部行单独计算一次。

**EXISTS谓词**

EXISTS谓词它的输入是一个子查询；如果子查询能够返回任何行，该谓词则返回TRUE，否则返回FALSE。

和其他谓词一样，我们可以使用NOT逻辑运算符来否定EXISTS。

性能优化方面

如果SQLServer引擎什么也没有找到，也足以决定子查询是否返回了。而无须处理所有满足条件的行。可以把这种处理方式看做是一种“短路（short-circuit）”，它能提高处理效率。

EXISTS另一个有趣的方面是：与T-SQL大多数谓词不同，EXISTS谓词使用的是二值逻辑，而不是三值逻辑。如果稍加考虑就能明白：不知道查询是否有返回行的情况是不存在的。

**子查询常见问题**

**NULL问题**

当对至少返回一个NULL值的子查询使用NOT IN谓词时，外部查询总会返回一个空集。因为NOT （FALSE OR UNKNOWN）的计算结果为UNKNOWN他会被过滤掉。

解决方案：

1. 在子查询中排除NULL值
2. 使用EXISTS谓词，因为EXISTS是二值逻辑，天生就能过滤UNKNOWN。所以使用NOT EXISTS比NOT IN安全。

**集合运算**

集合运算是对输入的两个或多个集合进行的运算，最终输出一个结果集。

T-SQL支持3种集合运算：并集（UNION），交集（INTERSECT）和差集（EXCEPT）。

集合运算的基本格式为：

*输入的集合1*

*<集合运算>*

*输入的集合2*

*[Order by …]*

需要注意的是，集合运算涉及的两个查询不能包含ORDER BY 子句。但可以为整个集合运算的结果集增加一个ORDER BY 子句。

参与集合运算的两个查询生成的结果集必须包含相同的列数，而且相应列必须具有兼容的数据类型。这里指的是优先级较低的数据类型必须能隐式地转换为较高的数据类型。

集合运算结果中的列名由第一个查询决定，因此如果要分配列名，应该在第一个查询中分配相应的列名。

集合运算还有一个特点就是它任务两个NULL值是相等的 。

**交并差集合运算**

UNION ALL集合运算返回在输入的多集中出现的所有行，它实际上不会对行进行比较，也不会删除重复行。

UNION （DISTINCT）集合通过删除重复记录，可以把两个输入的多集转变为一个集合。

从物理处理过程来看，SQLServer 不一定先删除输入多集中的重复行，在进行集合运算。相反，它可以先把两个多集组合到一起，然后再删除重复行。

运用场景：如果在集合运算中可能存在重复行，且要返回重复行，则使用UNION ALL。如果可能存在重复行，但是返回互不相同的行，则使用UNION。如果在输入两个集合中不可能出现重复行，推荐使用UNION ALL，因为可以避免SQLServer为检测重复行而带来的性能损失。

INTERSECT (DISTINCT)如果一个行在两个输入多集中都至少出现一次，那么交集返回的结果中将包含这一行。

EXCEPT (DISTINCT)集合运算是在逻辑上先删除输入多集中的重复行，然后返回只在一个集合中出现，在第二个集合中不出现的所有行。换句话说，一个行能够被返回，仅当这个行在第一个输入的多集中至少出现一次，而且在第二个集合中没有出现过。需要注意EXCEPT是不对称的。

**集合运算优先级**

INTERSECT的优先级比UNION和EXCEPT运算优先级高，而UNION和EXCEPT优先级相等。

在包含多个集合运算的查询中，优先计算INTERSECT然后，从左到右的顺序依次计算。

我们可以使用括号来提升到最大优先级!

**数据修改**

**INSERT语句**

INSERT VALUES语句可以将基于指定值的行插入表。

SQL Server2008增强了VALUES语句的功能，允许在一条语句中指定由逗号隔开的多行记录。例如：

INSERT INTO T\_Order

VALUES (1,1,'冰箱','2017-01-01'),

(2,1,'洗衣机','2017-01-01'),

(3,1,'电视','2017-01-01')

这个操作是原子操作，这意味这如果有任何一行报错，那么所有行都会插入失败。

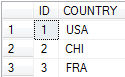
SQL Server2008还增强了VALUES子句的功能，现在可以用它来构建虚拟表（virtual table）。这种功能成为行值构造函数（row value contructor），或表值构造函数（table value contructor）。例如：

SELECT \* FROM

(VALUES(1,'USA'),(2,'CHI'),(3,'FRA'))

AS O(ID,COUNTRY)

它会生成如下表：



可以使用  INSERT INTO <target\_table> SELECT <columns> FROM <source\_table>  高效地将大量行从一个表（例如临时表）传输到按最小方式记录日志的其他表中。 按最小方式记录日志可以提高语句的性能，减少在事务期间此操作填充可用事务日志空间的可能性。

INSERT EXEC语句可以把存储过程或动态SQL批处理返回的结果及插入表中

SELECT \* INTO <target\_table> From <source\_table> 语句的作用是创建一个目标表，并用返回的结果填充它。

SELECT INTO它可以复制来源的表的基本结构（包括列名，数据结构，是否允许为NULL及IDENTITY属性）和数据。不过它不会复制索引，触发器，约束，如果需要这些东西，你可以手动添加。

SELECT INTO 的另一个优点是它会按照最小日志记录模式来执行操作，与完整日志模式相比，这种模式执行效率很高。

BULK INSERT语句用于将文件中的数据导入一个已经存在的表。

IDENTITY属性，被它标识的数字类型的列，能在插入数据时自动递增。

如果你想获取这个新生成的标识值（例如：你想获取新增的对象），可以查询@@IDENTITY或者SCOPE\_IDENTITY()，推荐使用SCOPE\_IDENTITY()因为它会返回当前作用域内会话生成的最后一个标识符，而@@IDENTITY则不会考虑。

**删除数据**

T-SQL提供两种从表中删除数据行的语句：DELETE和TRUNCATE

DELETE 语句是标准的SQL语句，它用于根据谓词删除表中的数据。

DELETE采用的是完整模式的日志处理，当删除大量数据时，可能会花费大量时间。

TRUNCATE不是标准的SQL语句，它用于删除表中所有数据，与DELETE不同的是，它采取的是最小模式记录日志，在性能上它们有很大差异。例如：删除几百万条数据时，使用TRUNCATE只需要几秒钟，而DELETE则需要几分钟甚至几小时。

当表中有标识列时，TRUNCATE会清除标识种子，而DELETE不会清除。

由于TRUNCATE语句执行速度非常快，因为有时具有一定的危险性。当清空表时搞错了要操作的表，类似的事情时有发生。

为避免发生这样的事故，可以简单地创建一个虚拟表（dummy table），以便于可以及时回滚。

T-SQL支持一种基于联接的语法，由于联接本身具有过滤的作用，因为它有一个基于谓词的过滤器（ON子句）。通过连接可以访问另一个表中相关的属性（列），并在WHERE子句中引用这些属性。

例如：

删除所有中国客户的订单

DELETE FROM O

FROM T\_Order AS O

JOIN T\_Customer AS C

ON O.CustID = C.CustID

WHERE C.Country = 'China'

也可以改成EXISTS+子查询的形式

DELETE FROM T\_Order

WHERE EXISTS(

SELECT \* FROM T\_Customer AS C

WHERE C.CustID = T\_Order.CustID AND C.Country = 'China'

)

**修改数据**

UPDATE语句是标准的SQL语句，用于对表中数据行的一个子集进行更新。为了标识作为更新的子集行，须要在WHERE子句中指定一个谓词。在SET子句中要指定更改的列和这些列的新值（或表达式），各列之间用逗号分隔。

“同时操作”是SQL表达式中一个重要的特性，在编写UPDATE语句时应该谨记这一点。

例如：

UPDATE T

SET col1 = col1 + 10,col2 = col1 + 10

假如在更新钱col1等于100，col2等于200，你能计算出最终的结果值吗？

如果不考虑同时操作的概念，你可能认为col将等于110，col2等于120。好像赋值表达式是从左到右执行的。但是两个表达式是同时执行的它们都等于110.

用同时操作的思想，如果想交换两个变量的值该怎么操作呢？在大多数编程语言中，得需要一个临时变量。然而在SQL中这个问题非常简单：

UPDATE T

SET col1 = col2,col2 = col1

和DELETE一样，T-SQL也支持一种基于联接的UPDATE语法，这不是标准的SQL语法。和基于联接的DELETE语句一样，联接在此也起到过滤的作用。

UPDATE还支持一种特殊语法，可以对表中的数据更新的同时为变量赋值，这种语法可以使你不必单独使用UPDATE和SELECT语句，就能完成同样的任务。

例如：

declare @i AS int = 123;

update T\_Customer

set @i = CustID = CustID + 1

select @i

**合并数据**

SQLServer2008引入了一个称为MERGE的语句，它在一条语句根据逻辑条件做出不同的修改操作（INSERT,UPDATE,DELETE），使用MERGE的好处是：用较少的代码就可以表达需求，提高查询性能，因为它可以更少地查询设计表。

在MERGE子句中指定目标表的名称，在USING子句中指定来源表的名称。然后在ON子句中用谓词来定义合并条件，合并条件用来定义来源表中哪些行在目标表中有匹配，哪些没有匹配。

我们可以在WHEN MATCHED THEN子句中定义找到匹配行时执行的操作，也可以在WHEN NOT MATCHED THEN子句中定义没有找到匹配行时要进行的操作。

例如：

--MERGE语句

MERGE T\_Employee AS e1

USING T\_Employee2 AS e2

ON e1.Empid = e2.Empid

WHEN Matched THEN --以USING的表为准，修改MERGE的表

UPDATE SET

e1.FirstName = e2.FirstName,e1.LastName = e2.LastName,e1.Country = e2.Country

WHEN NOT Matched THEN --以USING的表为准，新增MERGE的表

INSERT(FirstName,LastName,Country) VALUES

(e2.FirstName,e2.LastName,e2.Country)

WHEN NOT MATCHED BY SOURCE THEN --以USING的表为准，删除MERGE的表

UPDATE SET

e1.IsDelete = 1;

T-SQL还支持第三种子句：WHEN NOT MATCHED BY SOURCE它用于当来源表中没有行对应目标表时，一般可以采取删除操作。

我们还可以在WHEN MATCHED后面添加AND来为操作子句增加谓词条件。

**带有TOP的数据更新**

SQLServer2005新增了数据修改语句（INSERT,UPDATE,DELETE）中使用TOP选项功能。SQLServer2008则把TOP选项支持拓展到新增的MERGE语句中。

它的作用是，当已经处理的行达到指定数量或百分比后，会立即停止处理修改语句。可是与SELECT语句不同，不能为修改语句中的TOP语句指定ORDER BY子句。然而我们可以使用表表达式来解决这个问题。

例如：删除最后50条数据

WITH C AS

(

SELECT TOP 50 \*

FROM NUMS

ORDER BY n DESC

)

DELETE FROM C

将最后50条数据加50

WITH C AS

(

SELECT TOP 50 \*

FROM NUMS

ORDER BY n DESC

)

UPDATE C

SET n += 50

**事务与并发**

**事务**

事务是作为单个工作单元而执行的一系列操作，如查询和修改数据，甚至是修改数据的定义。

定义数据边界的方式有显式和隐式两种。显式事务的定义需要以BEGIN TRAN语句作为开始。如果想提交事务，则应该以COMMIT TRAN语句显式结束事务；如果不想提高事务，则以ROLLBACK TRAN语句显式结束事务。

如果不显式定义事务边界，SQLServer会默认把每一条语句作为一个事务。

事务必须有四种属性：原子性，一致性，隔离性，持久性。

原子性：在事务中进行的修改，要么全部执行，要么全不执行。如果在事务完成之前系统出现故障，SQLServer会撤销在事务中的修改。

一致性：为了事务在查询和修改时数据不发生冲突。

隔离性：隔离性是一种用于控制数据访问的机制，能够确保事务只能访问处于期望的一致性级别下的数据。SQLServer使用锁对各个事务之间正在修改和查询的数据进行隔离。

持久性：在将数据修改写入到磁盘之前，总是先把这些修改写入到事务日志中。这样子，即使数据还没有写入到磁盘中，也可以认为事务是持久化的。这是如果系统重新启动，SQL Server也会检查数据库日志，进行恢复处理。

**锁**

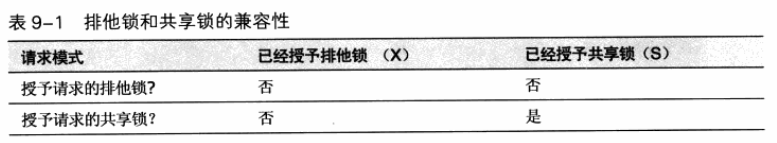
SQL Server使用锁来实施事务隔离属性。

锁是事务获取资源的一种控制资源，用于保护数据资源。放置其他事务对数据进行冲突或不兼容访问。

排他锁：当试图修改数据时，事务会为所依赖的数据资源请求排他锁。一旦授予，事务将一直持有排他锁，直至事务完成。

共享锁：当试图读取数据时，事务默认会为所依赖的数据资源请求共享锁，读操作一完成，就立即释放资源上的共享锁。这种模式下，多个事务可以同时持有同一资源上的共享锁。

事务之间的相互制约关系就是锁的兼容性。



用简单的话来总结一下事务之间锁的相互制约关系：如果数据正在由一个事务进行修改，其他事务就既不能修改也不能读取（至少默认不能）该数据，知道第一个事务完成。如果数据正由另一个事务读取，其他事务就不能修改该数据（至少默认不能），但可以读取。

可锁定资源的类型

SQL Server可以锁定不同类型或粒度的资源，这些资源类型包括Row ID或KEY(行)，PAGE(页)，对象(如表)，数据库等。行位于页中，而页则是包含在表或索引数据的物理数据块。

为了获得特定资源类型上的锁，事务必须先在更高粒度级别上获得相同模式的意向锁。例如，为了获得某一行上的排他锁，事务必须现在那一行的页上获得意向排他锁，并在包含那一页的数据对象上获得意向排他锁。

意向锁的目的是为了在较高的粒度级别上有效检测不兼容的锁定要求，防止不兼容的锁。



SQL Server动态决定应该锁定哪种类型的资源。自然，为了获得良好的并发性，最好是只锁定需要的资源，即只锁定受影响的哪些行。

**检测阻塞**

如果一个事务持有某一数据资源上的锁，而另一事务请求相同资源上不兼容的锁，则对新锁的请求将被阻塞，发出请求的事务进入等待状态。在默认情况下，被阻塞的请求会一直等待，直到原来的事务释放相关的锁。

如果事务的运行时间太长，会导致持有锁的时间也过久。这是可以尝试缩短事务处理，把不属于工作单元的操作移到事务外面。在某些情况下，应用程序的bug也会导致事务一直打开。

具体的SQL语句在SQL文件中

**隔离级别**

事隔离级别用于控制并发用户如何控制并发用户读写的操作。读操作可以是任何检索数据的语句，默认使用共享锁。写操作是指任何对表进行修改的语句，需要使用排他锁。

可以设置的隔离级别有6个：READ UNCOMMITTED(未提交读)，READ COMMITTED(可提交读)，REPEATABLE READ(可重复读)，SERIALIZABLE(可序列化)，SNAPSHOT(快照)，以及READ COMMITTED SNAPSHOT(已经提交隔离)。

**获取事务隔离级别(isolation level)**

DBCC USEROPTIONS

设置隔离级别

*SET TRANSACTION ISOLATION LEVEL <ISOLATION NAME>*

--注意:在设置回话隔离时(REPEATABLE READ)两个单词需要用空格间隔开,但是在表隔离中可以粘在一起(REPEATABLEREAD)

设置查询表隔离

SELECT ....FROM <TABLE> WITH (<ISOLATION NAME>)

隔离级别越高，一致性越高，并发性越低

**READ UNCOMMITTED(未提交读)**

未提交读是最低的隔离级别，在这个级别运行的事务，读操作不会请求共享锁。如果读操作不请求共享锁，就绝对不会和排他锁的写操作发生冲突。这意味这读操作也能读取未提交的修改（也称为**脏读**）。

**READ COMMITTED(已提交读)**

如果想避免读取未提交的修改，则需要使用要求更高的隔离级别。能够防止脏读的最低级别是READ COMMITTED，这也是SQL Server默认使用的隔离级别。这个隔离解绑只允许读取已经提价过的修改。它要求读操作必须获得共享锁才能进行，从而防止读取未提交的修改。

在该级别中，读操作一旦完成，就立即释放共享锁。他不会在事务持续期间保留它；这意味着在一个事务中间对相同数据资源的两次访问，没有共享锁会锁定该资源。因此，其他事务可以在两个读操作之间修改资源，读操作也有可能每次取到不同的值。这种现象称为**不可重复读**(non-repeatable read)或**不一致分析**(inconsistent analysis)。

**REPEATABLE READ(可重复读)**

如果想保证在事务内进行的两个读操作之间，其他任何事务都不能修改有当前事务读取的数据，则需要把隔离级别设置为REPEATABLE READ。在这种隔离级别下，事务中的读操作不但需要获得共享锁才能读取操作，而且获得的共享锁将一直保持到事务结束。

REPEATABLE READ隔离级别能够防止的另一种并发负面影响是**丢失更新**(lost update)，而较低的隔离级别不能防止此问题。丢失更新是指两个事务同时读取了同一个值，然后基于最初读取的值进行计算，接着再更新该值，这样最后提交的事务是“赢家”。覆盖其他事务所做的更新，这将导致事务丢失。

在REPEATABLE READ隔离级别下运行的事务，读操作会一直持有共享锁直到事务结束。但是事务只锁定第一个运行时找到的哪些数据资源，而不会锁定查询结果范围以外的其他行。因此，在同一事务进行第二次读取之前，如果其他事务插入了新行，而且新行也能满足读操作的查询过滤条件，那么这些新行也会出现第二次读取操作返回的结果中。这些新行称为**幻影**(phantom)，这种读操作称为**幻读**(phantom read)。

**SERIALIZABLE (可序列化)**

为了避免幻读，需要将隔离级别设置为SERIALIZABLE级别 ，它和REPEATABLE READ类似：即读操作需要获得共享锁才能读取数据，并保留共享锁直到事务结束。不过SERIALIZABLE隔离级别增加一个新内容-------逻辑上，这个隔离级别会让读操作锁定满足搜索条件的键整个范围。这就意味着读操作不仅锁定了满足条件的那些行，还锁定了未来可能满足搜索条件的行。

**死锁**

死锁(deadlock)是指进程间互相永久阻塞的状态，可能涉及两个或多个进程。例如：进程A阻塞了进程B，而进程B又阻塞了进程A。或者进程A阻塞了进程B，进程B阻塞了进程C，进程C阻塞了进程A。在任何一种情况下，SQL Server都可以检测阻塞，并选择终止其中一个事务以干预死锁状态。

除非指定了其他方式，SQL Server会选择终止做过操作最少的事务，因为这样可以让开销降低到最小。不过，用户也可以自己指定死锁情况下会话的优先级(DEADLOCK\_PRIORITY)。

那么我们怎么减少死锁呢？

显然，事务的处理事件越长，持有锁的事件就越长，死锁的可能性就越大。应该尽可能保持事务简短，在逻辑上将不属于同一工作单元的操作移到事务之外。

将两个事务按同样的顺序来访问资源，也可以避免死锁。

建立良好的索引支持也能避免死锁。例如：Connection1中对Order表中ID为1的进行操作，Connection2中对Order表中ID为2的进行操作，所以他们不应该产生冲突。但是如果在Order表中没有索引来支持查询筛选，SQL Server就必须进行全表扫描(并锁定)所有行。这样当然会造成死锁。

**可编程对象**

**变量**

用DECLARE声明一个或多个变量，用SET语句可以把一个变量设置成指定的值。

DECLARE @A AS VARCHAR(50);

SET @A = 1

SQL Server2008新增了对同一语句声明和赋值变量。

DECLARE @B AS INT = 10;

SET语句每次只能对一个变量进行操作，所以如果要对多个变量赋值，就必须使用多次SET语句。当从同一行取出不同的列的值时，可能会带来不必要的开销。

SET @A = (SELECT Name FROM T\_Order WHERE OrderID = 1)

SET @B = (SELECT OrderID FROM T\_Order WHERE OrderID = 1)

SQL Server还支持一种非标准的赋值SELECT语句，允许在单独的语句中既能查询数据，又能把同一行的多个值赋值给多个变量。

SELECT @A = Name,@B = OrderID

FROM T\_Order

WHERE OrderID = 1

*标量子查询：返回结果只有一行的子查询。*

**批处理**

批处理是客户端应用程序发送一条或多条T-SQL语句到SQL Server，SQL Server将批处理语句作为单个可执行的单元。批处理要处理的阶段有：分析，解析，优化。

SQL Server Management Studio提供了一个客户端命令GO，可以发出一批T-SQL语句结束的信号。注意，GO是客户端的工具，而不是T-SQL的服务器命令。

批处理是作为一个单元而进行分析和执行的一组命令。如果分析成功，SQL Server接着会尝试执行批处理。如果批处理中出现语法错误，整个批处理就不会提交到SQL Server执行。

变量是属于定义它们的批处理的局部变量，如果试图引用在其他批处理中定义的变量，SQL Server引擎会提示引用的变量还未定义。

DECLARE @A AS INT = 10;

PRINT @A;

GO

PRINT @A;

GO

不能在同一批处理和其他语句同时编译的语句：CREATE DEFAULT,CREATE FUNCTION,CREATE PROCEDURE,CREATE RULE,CREATE SCHEME,CREATE TRIGGER及CREATE VIEW。

IF OBJECT\_ID('V\_Shop','C') IS NOT NULL DROP TABLE V\_Shop

GO

CREATE VIEW V\_Shop AS

SELECT \* FROM T\_Shop

GO

如果在同一个批处理中既对数据对象进行了修改，又查询了对象，那么SQL Server可能还不知道架构发生了什么变化，因而无法执行数据处理语句，报告解析错误。这种问题的解决方式是把DDL语句和DML语句分隔到不同的批处理中。

SQL Server2005对Go这个客户端工具，进行了增强，让它可以支持一个正整数参数，表示Go之前的批处理将执行的次数。

CREATE TABLE #T(ID INT IDENTITY)

SET NOCOUNT ON;

INSERT INTO #T DEFAULT VALUES

GO 100

SELECT \* FROM #T

*刚才在判断视图是否存在时使用了，object\_id这是一个函数，用来返回数据库中对象的object\_id(对象标识符)的，如果不存在该对象，则返回NULL，存在返回一串数字。*

*我们常在建表，视图，存储过程中会用到，它有两个参数，一个是对象名，一个是类型(可选)，*

*常见的类型有：*

*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  
  
SO = 序列对象  
  
U = 表（用户定义类型）  
  
V = 视图*

**流程控制**

T-SQL提供了几种非常基础的流程控制元素，其中包括IF…ELSE和WHILE。

IF…ELSE元素用来根据条件来控制代码的执行流程。如果条件为TRUE，则执行指定的语句块；如果条件为FALSE或UNKNOWN则执行另一个语句块（这部分是可选的）。

如果需要在IF…ELSE部分运行多条语句，则可以使用语句块。语句块的边界是用一对BEGIN和END的关键字标识的。

**临时表**

有时需要把数据临时保存到表中，而且在某些情况下，你可能并不想使用永久性的表。例如，假设你需要让数据只对当前会话有效，或者甚至只对当前批处理有效。

在这种情况下，使用临时表会更方便，SQL Server支持三种临时表：局部临时表，全局临时表，表变量。

SELECT \*

INTO #T\_Order

FROM T\_Order

**局部临时表**

要创建局部临时表，只需要在命名时以单个数字符号（#）作为前缀，例如#T1。所有三种类型的临时表都是在tempdb数据库中创建的。

只有在创建会话的SQL Server实例断开时，SQL Server才会自动断开它。

SELECT \*

INTO ##T\_Order

FROM T\_Order

可以使用临时表的场合是：当你的处理需要把中间的结果集临时保存起来（如在一个循环中），以便以后的查询这些临时数据。

另一个查询是需要多次访问某个开销昂贵的处理结果。

**全局临时表**

如果创建的全局临时表，则它对其他所有会话可见。

当创建临时表的会话断开数据库的联接，而且也没有活动在引用全局临时表时，SQL Server会自动删除相应的全局临时表 。要创建临时表，只需要在命名时用（##）作为前缀，如##T1。

**表变量**

声明表变量的方式和声明其他变量类似，使用的都是DECLARE语句。

从性能上考虑，对于少量数据（只有几行），使用表变量更有意义，否则，应该使用临时表。

DECLARE @T\_Order table(

ID int,

Name varchar(50)

)

INSERT INTO @T\_Order(ID,Name)

SELECT OrderID,Name

FROM T\_Order

INSERT INTO @T\_Order(ID,Name)

VALUES(100,'微波炉')

**动态SQL**

SQL Server允许用字符串来动态构建T-SQL代码的一个批处理，接着再执行这个批处理。

这种功能称为动态SQL（dynamic SQL）。SQL Server提供了两种执行动态SQL的方法：

使用Exec命令和使用sp\_executesql存储过程。

常在“查询条件不固定”，“改善特定任务的性能”场景下应用。

**EXEC命令**

Exec接受一个字符串作为输入参数，执行里面的批处理代码。Exec及支持普通字符，也支持Unicode字符。

DECLARE @Msg AS varchar(100);

SET @Msg = 'PRINT ''HELLO WORLD''';

EXEC(@Msg)

**sp\_executesql存储过程**

Sp\_executesql既接受输入参数也接受输出参数，与Exec命令不同的是，他只支持Unicode字符。

Sp\_executesql存储过程有两个输入参数和一个参数赋值部分。在第一个参数@stmt中，需要指定包含想要执行的代码Unicode字符串。第二个参数@param是一个Unicode字符串，包含@stmt中所有输入，输出参数，接着为它们指定类型，用逗号分隔。

指定参数更有助于写出更安全和更有效的代码，就可以不必受SQL注入的困扰了。

DECLARE @sql AS nvarchar(100) = N'SELECT \* FROM T\_Order Where OrderID > @OrderID And CustID = @CustID'

-- And CustID = @CustID

exec sp\_executesql

@stmt = @sql,

@params = N'@OrderID AS INT,@CustID AS INT',

@OrderID = 1,

@CustID = 1

Sp\_executesql存储过程在性能上也比EXEC命令更好，因为它的参数化有助于重用缓存过的执行计划。

*执行计划：执行计划是SQL Server为查询生成的物理处理计划，包含了一组指令，说明要访问哪些对象，以什么顺序，使用哪个索引，如何访问它们，使用什么链接算法等等。*

**例程**

例程（routine）是为了计算结果或执行任务而对代码进行封装的一种编程对象。SQL Server支持三种例程：用户定义函数，存储过程，触发器。

**用户定义函数**

用户定义函数（UDF，user-defined function）的目的是要封装计算的逻辑处理，有可能需要基于输入的参数，并返回结果。

SQL Server支持两种用户定义函数：标量UDF和表值UDF。标量UDF只返回单个数据值。而表值UDF则返回一个表。使用UDF的优点之一是可以在查询中集成UDF。对于查询中返回单个值的表达式，在其出现的位置上，也能够使用标量UDF。表值UDF只能在查询的FROM子句中出现。

UDF不允许有任何副作用，副作用的含义是指不能对数据库中任何架构或者数据进行修改。

**存储过程**

存储过程是封装了T-SQL代码的服务端例程。存储过程可以有输入和输出参数，也可范湖多个查询的结果集。通过存储过程不但可以对数据进行修改，也可以对数据库架构进行修改。

和普通代码相比，存储过程有以下好处：

1. 存储过程可以封装逻辑处理。只需要修改数据库里的存储过程，所有用户都能使用修改后的版本。
2. 存错过程可以提高执行效率。存储过程默认情况下是重用执行计划的，而SQL Server对其他特殊执行计划有更多限制。另一个好处是减少网络的传输流量。客户端只需要向数据库发送存储过程的名称和参数。服务器就换处理所有代码，向调用者只返回输出结果。
3. 通过存储过程可以有更好的安全性
4. 在存储过程中可以整合所有的错误处理

**错误处理**

运行错误处理的主要工具是一种称之为**TRY…CATCH**的结构。通常我们是将T-SQL语句放入TRY块中（放在BEGIN TRY和END TRY关键字中间），而把错误处理代码放在CATCH块中（在BEGIN CATCH和END CATCH关键字中间）。

SQL Server中有一组函数来反馈有关错误的信息。**ERROR\_NUMBER**函数返回一个整数表示错误号，**ERROR\_MESSAGE**函数返回错误文本。我们可以对照sys.messages视图来查看所有的对照表。ERROR\_SEVERITY和ERROR\_STATE返回错误严重级别和状态号。ERROR\_LINE返回错误行号。最后ERROR\_PROCEDURE返回发生错误的存储过程或触发器的名称。

以下是封装好的一个收集错误的存储过程，这样所有的错误处理代码只需要在这里维护一次即可，只要在CATCH块中调用它，就行了。

CREATE PROCEDURE P\_ErrorMessage

AS

BEGIN

SET NOCOUNT ON;

print 'ERROR\_NUMBER:' + CAST(error\_number() AS VARCHAR(10));

PRINT 'ERROR\_MESSAGE:' + error\_message();

print 'ERROR\_SEVERITY:' + CAST(error\_severity() AS VARCHAR(10));

print 'ERROR\_STATE:' + CAST(error\_state() AS VARCHAR(10));

print 'ERROR\_LINE:' + CAST(error\_line() AS VARCHAR(10));

PRINT 'ERROR\_PROC:' + COALESCE(error\_procedure(),'not within proc')

END

GO

***COALESCE表达式***

返回参数中从左到右，第一个不为NULL的值，参数不限制。返回的数据类型为参数中最高优先级的类型。

**比较COALESCE和CASE**

COALESCE表达式是CASE表达式的语法糖，查询优化器会将

SELECT COALESCE(EXPRESSION1,EXPRESSION2,...)

转成如下的CASE表达式：

SELECT CASE WHEN EXPRESSION1 IS NULL THEN EXPRESSION1,

CASE WHEN EXPRESSION2 IS NULL THEN EXPRESSION2,

...

**比较 COALESCE 和 ISNULL**

1. 因为 ISNULL 是函数，它只能被计算一次。 如上所述，可以多次计算 COALESCE 表达式的输入值。
2. 确定结果表达式的数据类型方式不同。 ISNULL 使用第一个参数的数据类型，COALESCE 则遵循 CASE 表达式规则并返回具有最高优先级的值的数据类型。
3. ISNULL 只取 2 个参数而COALESCE 可取不同数目的参数。

*ps: ISNULL的第一个参数为被检查是否为NULL的表达式，第二个参数是当第一个参数为NULL时返回的值，需要注意的是，两个参数的类型必须相同*

**字符串聚合操作**

常常有一种需求就是行转列，比如要将

|  |  |
| --- | --- |
| Empid | Name |
| 1 | Jack |
| 1 | Lucy |
| 2 | Jim |
| 2 | Alax |

转成

|  |  |
| --- | --- |
| Empid | Name |
| 1 | Jack,Lucy |
| 2 | Jim,Alax |

在DB2中我们有现成的函数可以实现这个功能，它就是LISTAGG。

SELECT EmpID,LISTAGG(NAME,',') FROM ORDERS

GROUP BY EmpID

之后我在网上查了下，SQL Server有没有类似的函数呢？

网上有很多种方法可以实现这个功能，比如将For xml path()和stuff配合使用，就能达到这个效果。

SELECT EMPID,value = STUFF(

(SELECT ','+NAME

FROM ORDERS as a

WHERE a.EMPID = b.EMPID

FOR XML PATH('')),1,1,''

)

from orders as b

group by EMPID