**数据库性能概要规范**

**Newegg.com**

***Document Information***

Last Updated 6/2/11

File name: 数据库性能概要规范.doc

***Author Information***

Author David Dong

Group MIS

**目录**

[**1** **目的** 5](#_Toc120163302)

[**2** **Performance Audit Checklist** 5](#_Toc120163303)

[**3** **常用的数据库优化工具** 6](#_Toc120163304)

[**4** **性能调整** 6](#_Toc120163305)

[**4.1** **TSQL** 6](#_Toc120163306)

[**4.1.1** **不要访问多于你需要的数据** 6](#_Toc120163307)

[**4.1.2** **所有者.对象名（objectownername.objectname）** 6](#_Toc120163308)

[**4.1.3** **Union &Union All** 6](#_Toc120163309)

[**4.1.4** **Union & Jion** 8](#_Toc120163310)

[**4.1.5** **Distinct** 9](#_Toc120163311)

[**4.1.6** **TOP N** 10](#_Toc120163312)

[**4.1.7** **In And Exist** 11](#_Toc120163313)

[**4.1.8** **In and Between** 11](#_Toc120163314)

[**4.1.9** **Like** 11](#_Toc120163315)

[**4.1.10** **OR** 12](#_Toc120163316)

[**4.1.11** **Order by/ sorting** 12](#_Toc120163317)

[**4.1.12** **Exist and count(\*)** 13](#_Toc120163318)

[**4.1.13** **使用Case** 14](#_Toc120163319)

[**4.1.14** **Null Vs NotNull** 15](#_Toc120163320)

[**4.1.15** **Bit** 15](#_Toc120163321)

[**4.1.16** **字符串操作** 15](#_Toc120163322)

[**4.1.17** **在Where语句中存在冗余条件** 15](#_Toc120163323)

[**4.1.18** **在WHERE子句中的列上使用函数** 16](#_Toc120163324)

[**4.1.19** **避免在WHERE中使用no- sargable** 16](#_Toc120163325)

[**4.1.20** **避免使用游标** 17](#_Toc120163326)

[**4.1.21** **使用存储过程** 25](#_Toc120163327)

[**4.1.22** **不要包括不做任何事情的代码.** 26](#_Toc120163328)

[**4.1.23** **sp\_?** 26](#_Toc120163329)

[**4.1.24** **CHECKSUM索引** 26](#_Toc120163330)

[**4.1.25** **索引提示（Index hint）** 27](#_Toc120163331)

[**4.2** **事务和死锁** 27](#_Toc120163332)

[**4.2.1** **Transaction** 27](#_Toc120163333)

[**4.2.2** **Complex transaction** 28](#_Toc120163334)

[**4.2.3** **Deadlock** 29](#_Toc120163335)

[**4.3** **表操作** 31](#_Toc120163336)

[**4.3.1** **Avoid Table scan** 31](#_Toc120163337)

[**4.3.2** **SELECT INTO** 31](#_Toc120163338)

[**4.3.3** **Update** 31](#_Toc120163339)

[**4.3.4** **大批量更新，删除，插入数据** 31](#_Toc120163340)

[**4.3.5** **TempTable & Derived Tables** 32](#_Toc120163341)

[**4.3.6** **有效的使用JOIN** 35](#_Toc120163342)

[**4.3.7** **父子表** 35](#_Toc120163343)

[**4.3.8** **关于统计信息（Statistics）** 36](#_Toc120163344)

[**4.3.9** **为列选择合适数据类型** 37](#_Toc120163345)

[**5** **参考资料** 37](#_Toc120163346)

**版本控制**

修改历史

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **日期** | **描述** | **修改者** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. **目的**

通过对一些常见的影响数据库性能的查询部分进行分析，提出相应的解决方法，以提高公司业务数据库业务系统的高可用性及可并发性。

本文基本分为两部分，第一部分会给出一个简单的checklist,列出了常见的一些影响性能的问题，第二部分则强调在设计，开发方面的性能调整问题。

另外，由于个人经验的关系，本文档可能会有很多的缺陷，如果各位能够提供相关不足，将非常感激,请发送邮件至[DavidDong@newegg.com](mailto:DavidDong@newegg.com)

***说明：***

在不同的数据库环境中，相同的数据库性能设置技巧可能会有完全不同的表现结果，所以性能方面的技巧一般都是与特定的环境相关的，不能盲目的相信所谓的性能调整技巧，而在我们每次进行相关性能调整的过程中，我们也需要进行多种测试来达到最理想的效果。

1. **Performance Audit Checklist**

|  |  |
| --- | --- |
| **Transact-SQL Checklist** | **Your Check** |
| 查询时不返回多余的数据(比如没有SELECT \*) |  |
| 尽量不使用游标 |  |
| 如果确实需要使用游标，能否有其他的解决方法 |  |
| 注意T-SQL语句中UNION 和 UNION ALL区别 |  |
| 尽量避免使用SELECT DISTINCT |  |
| 注意T-SQL语句中Exist() 和 Count(\*)区别 |  |
| 注意T-SQL语句中IN 和 EXISTS区别 |  |
| 在Where语句中不存在no- sargable（"IS NULL", "<>", "!=", "!>", "!<", "NOT", "NOT IN", "NOT LIKE", 和 "LIKE '%500'") |  |
| 注意在WHERE子句中不存在数据类型的转换 |  |
| 注意在WHERE语句中不存在对列进行函数操作 |  |
| 注意查询中尽力避免使用临时表 |  |
| 注意到临时表不可由derived table替代 |  |
| 注意到查询使用了合适的表提示 |  |
| 注意到没有以sp\_作为前缀的存储过程 |  |
| 注意到存储过程中是否使用了SET NOCOUNT ON |  |
| 注意到所有的数据库对象是依dbo.objectname的形式来引用 |  |
| 尽力保证事务尽可能短小并尽可能不使用嵌套事务 |  |
| 对JOIN操作进行了优化 |  |

1. **常用的数据库优化工具**

* **Query Analyzer**
* **Profiler**
* **Index Wizard**
* **System (Performance) Monitor**

1. **性能调整**

以下分别从TSQL、事务和死锁、表操作这三个方面来讲性能调整的问题。

1. **TSQL**
2. **不要访问多于你需要的数据**

这个听起来是多余的，但是确实是非常必要的，做起来也不大容易。其中包括不要返回给客户端不需要的列和行。比如在SELECT语句中不要使用SELECT \*，否则会经常返回给客气端多于它们所需要的数据，这样可以有效减轻网络传输压力，减少不必要的I/O，以及减少内存耗费，并能够使查询优化器最优化我们的查询执行计划，从而减少潜在的性能问题。

1. **所有者.对象名（objectownername.objectname）**

建议在所有对表、视图和存储过程引用时，加上它们的前缀。下面讨论一下为什么使用前缀可以改进查询性能。

***例子：***有一存储过程dbo.foo，此存储过程运行查询

SELECT col1 FROM table1

当用户Lucy调用存储过程时，查询优化器必须决定是检索Lucy.table1还是检索dbo.table1。然后，当用户Lily调用同一个存储过程时，查询优化器必须对查询计划进行重新编译，以决定用户是需要Lily.table1还是需要dbo.table1。 但是如果把上面的SELECT语句修改如下

SELECT col1 FROM dbo.table1

查询优化器将不会遇到任何模糊性，从而避免重新编译，达到提升性能的目的。

1. **Union &Union All**

首先讲一下UNION的工作原理，当使用UNION语句时，它的功能与在结果集上SELECT DISTINCT类似，也就是说使用UNION时它会首先合并两个结果集，然后执行一个类似于SELECT DISTINCT的操作，以避免重复行的出现。这个过程在两上结果集没有任何重复行的情况下也会进行DISTINCT处理，所以如果我们确认在UNION的两个结果集确实存在重复行，并且要消除重复行的出现时，就可以使用UNION.从另一方面来说，如果我们知道在结果集中并不会存在重复行，或者说出现重复行对我们的应用程序没有什么影响时，我们应该使用UNION ALL语句来替代 UNION语句，UNION ALL和UNION相比的优点在于它并不会对两个结果集进行SELECT DISTINCT操作，这样可以节省SQL Server 的资源使用。

***例子***：比较UNION和UNION ALL的差别：

--query 1,use Union All,the second result will apend to the first result set .

SELECT 1 ,2

UNION ALL

SELECT 1,2

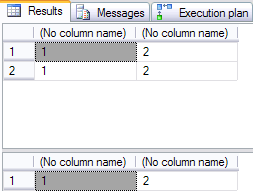
--query2,use union,this will do the distinct operation

SELECT 1 ,2

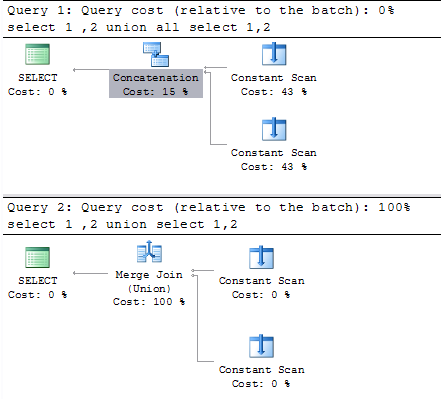
UNION

SELECT 1,2

我们来看一下其结果集，可以在第一个查询中只是把第二个查询的结果集附加到第一个结果集中，而第二个查询就执行了DISTINCT操作，消除了重复行：



下面来看性能方面的对比，分析下面的执行计划，我们可以看到第二个操作非常消耗资源，如果两个查询一块执行时，第二个查询几乎使用了所有的数据库资源（主要消耗在MERGE JOIN中的消除重复行部分）：



结论：在对UNION和UNION ALL进行选择时，除非必要，推荐使用UNION ALL

1. **Union & Jion**

在比较UNION和UNION ALL的基础上，下面来看UNION合并结果集和使用JION查询得到结果集之间的差别。

***例子：***使用UNION来合并多个结果集的情况：

USE PUBS;

GO

SELECT \* FROM DBO.SALES

WHERE QTY<10

UNION

SELECT \* FROM DBO.SALES

WHERE QTY>11

上面的查询可以被重写为下面的查询，这样一般就会有性能提升：

USE PUBS;

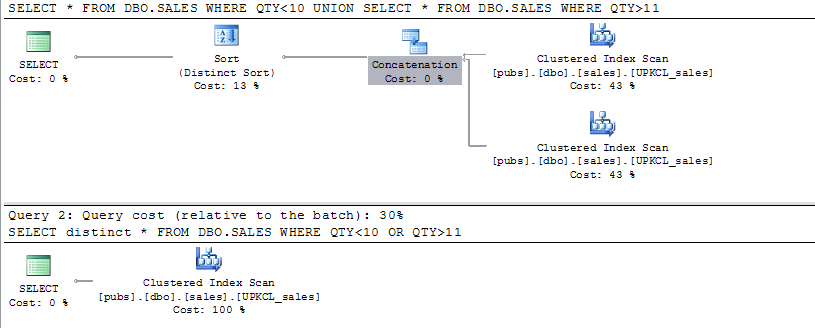
GO

SELECT distinct \* FROM DBO.SALES

WHERE QTY<10 OR QTY>11

***NOTE：***如果知道结果集中并不存在重复的数据行，我们也可以使用UNION ALL来提高性能。但是Union All性能一般来说比Union要好，但是比Join要差。

查看它们的运行结果，可以发现运行结果是相同的，现在来分析一下它们的执行计划和性能差别：



可见使用UNION时执行两次CLUSTERED索引的扫描，并且要把结果使用DISTINCT合并起来，而第二个查询只要进行一次CLUSTERED索引进行扫描，然后直接展现出来，从而大大提高了性能。

但是如果能够使用JOIN达到目的的话，我们建议不要使用Union 和Union ALL而是直接使用JOIN来取我们需要的数据。

1. **Distinct**

有开发人员在写查询的时候不管是否需要都习惯性的在查询中写上DISTINCT,这是一个很不好的习惯，特别是当我们的查询中包含很大数据量的时候更会大大消耗有限的数据库资源并降低的应用程序性能。

DISTINCT应该在确认结果集中存在重复行，并且这些重复的确数据并不是我们所需要的数据时才能够使用。这是因为DISTINCT在数据库上执行了很多额外的操作从而消耗了很多数据库资源，这样会减少别的查询在执行时所能够使用的资源，增加数据库出现性能问题的几率。

***例子 ：***

USE PUBS;

GO

SELECT DISTINCT

au\_fname,

au\_lname

FROM authors

这是一个非常简单的SELECT DISTINCT查询语句的示例，但是当我们执行一个复杂的语句时就应该考虑重新编码以达到性能的要求，如下面的语句，当返回authore表中已经出版过书的作者时，很多人会像下面这样写：

SELECT DISTINCT

au\_fname,

au\_lname

FROM authors a JOIN titleAuthor t

ON t.au\_id = a.au\_id

但这时只想得到作者名字，我们可以重写这个语句来提升我们查询语句的性能：

SELECT au\_fname,

au\_lname

FROM authors a

WHERE EXISTS (

SELECT \*

FROM titleAuthor t

WHERE t.au\_id = a.au\_id

)

上面的查询能够提升性能是因为如果某个作者出过很多书，当查到这个作者出版的第一条记录时，就会停止对这个作者的处理。

我们要慎重考虑是不是真的需要DISTINCT

1. **TOP N**

如果我们的应用程序要返回上千行乃至上万行数据的时候,我们要考虑是不是真正的需要这么多数据，是不是可以使用TOP 操作符来限制返回给客户端的行数或者返回给客户端结果集行的百分比，这样可以减少资源的使用，提高数据库性能并且有效节省带宽

下面说一下关于减少网络传输压力的内容。如果我们每次多返回给客户端10行数据，每行多返回200个字节，每天10000次被执行，这就是一笔不小的网络传输成本(10\*200\*10000Byte)，另外，如果再加上传输图片等，将要占用更多的带宽。

***例子：***

USE PUBS;

GO

SELECT TOP 1 FNAME,LNAME,HIRE\_DATE FROM DBO.EMPLOYEE

WHERE LNAME LIKE 'A%'

这时如果有100,000行数据符合WHERE条件，也只返回限制的1行结果集。因为SQL Server在处理时，当结果集的行数达到TOP中指定的数目时，所有的处理都将停止，这样就可以潜在的提高SQL Server的负载，增加性能。

此外，TOP操作还可以让我们指定返回给客户端结果集行数的百分比，如:

USE PUBS;

GO

SELECT TOP 1 PERCENT FNAME,LNAME,HIRE\_DATE FROM DBO.EMPLOYEE

WHERE LNAME LIKE 'A%'

***NOTE：***如果一个 SELECT 语句既包含 TOP 又包含 ORDER BY 子句，那么返回的行将会从排序好的结果集中选择，并且只返回已排好序结果集的前 n 行（或者前百分之N行）。

1. **In And Exist**

当我们能够在我们的查询中选择使用IN和Exist语句时，推荐使用EXISTS，因为EXISTS一般更加高效(EXIST只遇到附合条件的很第一个结果时，此记录就退出处理)。

1. **In and Between**

在查询语句中选择使用IN和BETWEEN时，建议使用BETWEEN,因为它在很多场合下更高效。

***例子：***有一个customer表，在customer\_number列上有一个非聚集索引，进行如下查询：

SELECT customer\_number, customer\_name

FROM customer

WHERE customer\_number in (1000, 1001, 1002, 1003, 1004)

查询优化器可以使我们下面使用BETWEEN的语句比使用IN的语句性能更高效:

SELECT customer\_number, customer\_name

FROM customer

WHERE customer\_number BETWEEN 1000 and 1004

1. **Like**

当在WHERE中使用Like时，尽可能的在Like语句中使用一个或者多个前导字符，比如：

使用

LIKE 'm%'

而不是:

LIKE '%m'

如果在Like中使用一个前导字符，此时查询优化器就会自动使用相应索引（如果有合适索引存在的话）来处理这个查询，但是如果我们Like语句中的前导字符使用通配符，查询优化器不会使用索引，而会对表进行扫描，从而大大降低性能。这点在SQL Server 2005上有所提升，在SQL Server 2005中，查询优化器可以为类似于LIKE '%m' 语句使用索引。

**NOTE:**使用的前导字符越多，查询优化器就越有可能找到合适的索引，从而加快查询。

另外，如果应用程序的查询中有很多对CHAR和VARCHAR列进行Like操作时，我们可能要考虑使用SQL Server 的全文索引，全文索引对于处理这方面查询会大大提升性能。

1. **OR**

如果一个查询语句中包括多个OR子句，为提升性能，一般情况下可以重写为一系列查询并使用UNION ALL来合并结果集。

***例子：***

SELECT employeeID, firstname, lastname

FROM names

WHERE dept = 'prod' or city = 'Orlando' or division = 'food'

这个语句在WHERE中有三个独立的条件，为了使用到索引，我们要在这三列上创建组合索引。像这样的语句可能被重写为多个查询并使用UNION ALL来合并结果集，从而替代OR操作。如下：

SELECT employeeID, firstname, lastname FROM names WHERE dept = 'prod'

UNION ALL

SELECT employeeID, firstname, lastname FROM names WHERE city = 'Orlando'

UNION ALL

SELECT employeeID, firstname, lastname FROM names WHERE division = 'food'

每一个查询都得出一样的结果集（元数据相同），如果在这个表中只有dept列上存在索引，此时第一个SELECT就会使用此索引，而第二个第三个SELECT语句会进行表的扫描，而不是全部进行表的扫描。

这只是一个非常简单的例子，只是示范了如何重写查询来提高性能，如果这个查询比较复杂，那么使用UNION ALL就会大大提高性能，因为这样我们可以对每个语句进行调整，而如果是全在一个OR子句中，这些我们是做不到的。   
NOTE:注意在这里我们是使用UNION ALL而不是UNION，原因是UNION不光会对结果集进行合并，也会对结果集进行排序并移除所有的重复行，所以性能会大大降低。

1. **Order by/ sorting**

ORDER BY要占用很多额外的资源，所以除非在真正需要的情况下，否则不要在SELECT语句中使用ORDER BY。我们可以考虑，这些数据在客户端进行排序是否会更好，或者客户端是否需要我们对结果集进行排序。

SORT 一般是伴随着以下T-SQL 语句出现：

* + ORDER BY
  + GROUP BY
  + SELECT DISTINCT
  + UNION
  + CREATE INDEX （此时也需要对数据进行排序）

一般情况下，这些命令不可能避免，但是另一方面，我们可能想办法减少这些语句对资源的使用，这包括：

* + 尽可能减少被排序的行数,就是只对必要排序的数据行集进行排序。
  + 尽可能减少结果集中列的数目。
  + 尽可能减少所有行的物理宽度.
  + 尽可能对INT的列进行排序，而不是Character或者Char列。

当使用SORT操作时，一定要考虑上面的建议，并保持下面的原则 “除非必要，否则不要来对结果集进行排序”。

1. **Exist and count(\*)**

对数据库进行更新时，很多时候要首先判断被更新的记录是否存在，或者对删除对象之前，判断对象是否存在，此时不要使用SELECT COUNT(\*) 来标识，因为它的性能是非常低的。为了提升性能，我们可以使用IF EXISTS来实现相关的操作。IF EXISTS能提高性能是因为当有一条记录为真是，处理就会立即退出，但是COUNT(\*)会检查表中的每一条记录的情况，不管符合条件的数据有一条还是上万条。

***例子：***

USE PUBS

--COUNT(\*):

IF (SELECT COUNT(\*) FROM DBO.AUTHORS )>0

BEGIN

PRINT 'ASDF'

END

--下面是使用EXIST 的语句，也是性能较好的语句：

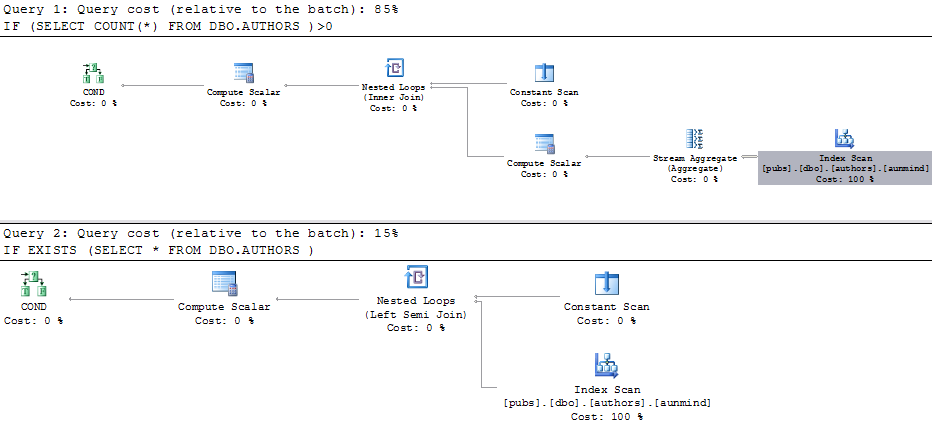
IF EXISTS (SELECT \* FROM DBO.AUTHORS )

BEGIN

PRINT 'ASDF'

END

看一下执行计划，可见使用EXIST确实能够提高性能。



虽然这两个查询语句都使用了一样的索引，但是在COUNT(\*)时返回的行的数据是表中符合条件的所有行，然后才对这些行进行Aggregate，这也就是为什么数据越多，使用COUNT(\*)性能越低的原因。

1. **使用Case**

当对一个表进行多次更新时，尽量合并到一个更新语句中。

***例子：***如果要对PUBS数据库中的相关书根据不同的条件来调整它的销售价格，需要调高商务类书的销售价格至原来的130%，调高非商务类书的销售价格至原来的110%，我们很多人在UPDATE的时候是以一个事务中包括两个独立的UPDATE语句来进行的，如下：

BEGIN TRAN

UPDATE titles SET price =price \* 1.30 WHERE type = 'business'

UPDATE titles SET price =price \* 1.10 WHERE type <> 'business'

COMMIT tran

上面语句的缺点是它会对表进行两次的读写，下面看一下如何把它们写到一个语句中来：

UPDATE titles

SET price =

CASE

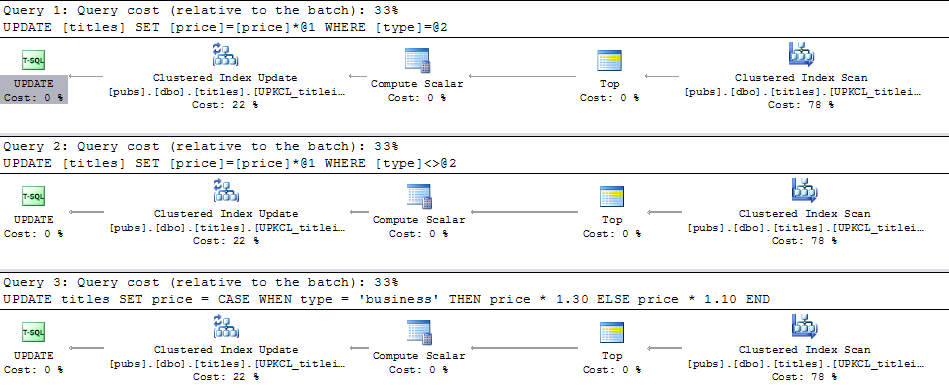
WHEN type = 'business'

THEN price \* 1.30

ELSE price \* 1.10

END

比较一下下面的执行计划，发现第一个查询语句的COST为66.7%,第二个查询语句的COST为33.3%.这样就可以大大提高性能。



1. **Null Vs NotNull**

在创建表时应该为每列创建默认值而不是允许为NULL,因为对可空列进行操作和对非可空列进行操作相比将需要更多额外的资源。如比较（comparisons）就是一个很好的例子。

所以我们应该尽可能的使用NOT NULL来替代NULL。

1. **Bit**

不推荐使用Bit列。

如果已经使用Bit列，请确保不要允许为NULL,如果此列有两个以上的状态，请考虑其它类型。

1. **字符串操作**

尽量避免在**Transact-SQL**中进行字符的串联操作，因为这是个很慢的操作，会造成应用程序的减慢(TSQL是为集合操作而优化的)。

1. **在Where语句中存在冗余条件**

在WHERE语句中小心地使用OR，这经常会得到一些多余的数据，同时也很容易在WHERE语句中出现冗余的子句。

***例子：***

SELECT companyid, plantid, formulaid

FROM batchrecords

WHERE （companyid = '0001' and plantid = '0202' and formulaid = '39988773'）

OR

（companyid = '0001' and plantid = '0202' ）

这儿的WHERE语句是冗余的:

companyid = '0001' and plantid = '0202' and formulaid = '39988773'

是下面子句的子集:

companyid = '0001' and plantid = '0202'

换句话来说，这个查询是冗余的，但SQL Server 2000的查询优化器并不会自动探测是否存在冗余（这个在SQL Server 2005中有所提高），它完全按你所写的条件去执行，此时SQL Server 就会首先得到所有的数据，然后使用SELECT DISTINCT来移除冗余的数据。如果删除以下的查询子句：

OR

companyid = '0001' and plantid = '0202'

此时这个查询语句就会加快很多。

1. **在WHERE子句中的列上使用函数**

默认情况下，在进行比较时，经常写类似下面的语句:

SELECT column\_name FROM table\_name

WHERE LOWER(column\_name) = 'name'

也就是说，大家认为SQL Server是大小写敏感的，但一般情况下我们SQL Server的配置为大小写不敏感，所以不使用类似于上面的语句，就可以大大加快查询的速度。

但如果数据库确实是配置为大小写敏感时，上面的代码依然性能很低。因为原则是：不要在WHERE中对列进行任何函数操作，在列上进行函数操作会导致此查询并不能够使用索引，如下：

SELECT member\_number, first\_name, last\_name

FROM members

WHERE DATEDIFF(yy,datofbirth,GETDATE()) > 21

如果需要进行函数操作的话可以写类似于下面的代码，下面的代码可以使用dateofbirth列上的索引：

SELECT member\_number, first\_name, last\_name

FROM members

WHERE dateofbirth < DATEADD(yy,-21,GETDATE())

1. **避免在WHERE中使用no- sargable**

像"IS NULL", "<>", "!=", "!>", "!<", "NOT", "NOT EXISTS", "NOT IN", "NOT LIKE",  和 "LIKE '%500'"的操作一般来说将导致查询优化器不能够使用索引来进行数据的查找。一般情况下，我们可以重写这样的语句来达到性能的提升。

***例子：***

下面语句：

WHERE SUBSTRING(firstname,1,1) = 'm'

可能被重写为下面的查询：

WHERE firstname like 'm%'

NOT其实并不是no-sargable，但是一般情况下尽可能在查询中查询中不使用NOT操作符，如：

WHERE NOT column\_name > 5

可以被重写为：

WHERE column\_name <= 5

1. **避免使用游标**

游标可以严重影响SQL Server性能，但是一些场合下不可避免，不过更多场合下我们是可以不使用游标的。所以如果现在你的应用程序使用T-SQL游标，请重新检查你的代码并尽可能的重写它们。

Transact-SQL是设计用来处理集合操作的，而不是对独立的数据记录行进行操作。不过SQL提供了一种对数据行进行操作的选项，在Transact-SQL中游标就可以被用来处理独立的行，但是问题就是它的操作是如此慢，理想情况下，在高性能的基于SQL Server 的应用程序中，游标应该被避免。

所以如果你要进行一个基于行的操作，设法找到其它的方法来完成这项工作是一个非常好的开发习惯。考虑使用以下选项来替代游标：

* **使用临时表**
* **使用WHILE循环**
* **使用 derived tables**
* **使用子查询**
* **使用CASE语句**
* **使用多个查询**

上面的这些选项有时候可以替代游标，从而大大提升性能。但并不是所有的情况下都能够使用以上的选项来替代游标，如果发现不可能避免使用游标，请参考下面的建议或许会有帮助：

* SQL Server提供了几种不同种类的游标，它们之间性能也存在着差异。我们要选择对服务器性能影响最小的能够达到目标的游标。最高效的游标就是fast forward-only游标。这个在我们的命名规范中也有提及
* 当使用游标时，取尽可能少的数据
* 当用过游标后，不仅CLOSE它，同时也要DEALLOCATE此游标，也就是说游标占用的存储单元也需要释放，如果只是简单的关闭而不释放，虽然此游标锁死的资源被释放，但SQL Server 的资源并没有释放，游标使用的资源依然存在直到你释放它为止。

SQL Server游标非常消耗数据库资源，这是因为它极低的性能所致。但是根据“存在就是必然”的原则(Montaque语录)，它们也有存在的道理，因为在有些地方是确实非常有用的，它们非常灵活，可以使我们对每行的数据进行非常灵活的操作，但一般情况下我们使用的游标可能使用其它方法来代替，比如Derived Table，集合查询和临时表，下面会讨论一下这个问题。

下面讨论游标的可选替代方案。

这里通过一些示例来演示一下如何不使用游标来解决一些以前用游标来解决的问题。

***例子：***

首先，先看一下简单的游标，此游标在循环一个表，然后我们看一下如何不使用游标来达到相同的目标。

使用游标：

if exists (select \* from sysobjects where name = N'prcCursorExample')

drop procedure prcCursorExample

go

CREATE PROCEDURE prcCursorExample

AS

/\*

\*\*put your comment here

\*\*

\*/

SET NOCOUNT ON

-- declare all variables!

DECLARE @iRowId int,

@vchCustomerName nvarchar(255),

@vchCustomerNmbr nvarchar(10)

-- declare the cursor

DECLARE Customer CURSOR FOR

SELECT iRowId,

vchCustomerNmbr,

vchCustomerName

FROM CustomerTable

OPEN Customer

FETCH Customer INTO @iRowId,

@vchCustomerNmbr,

@vchCustomerName

-- start the main processing loop.

WHILE @@Fetch\_Status = 0

BEGIN

-- This is where you perform your detailed row-by-row

-- processing.

-- Get the next row.

FETCH Customer INTO @iRowId,

@vchCustomerNmbr,

@vchCustomerName

END

CLOSE Customer

DEALLOCATE Customer

RETURN

不使用游标：

if exists (select \* from sysobjects where name = N'prcLoopExample')

drop procedure prcLoopExample

go

CREATE PROCEDURE prcLoopExample

AS

/\*

\*\*put your comment here

\*/

SET NOCOUNT ON

-- declare all variables!

DECLARE @iReturnCode int,

@iNextRowId int,

@iCurrentRowId int,

@iLoopControl int,

@vchCustomerName nvarchar(255),

@vchCustomerNmbr nvarchar(10)

@chProductNumber nchar(30)

-- Initialize variables!

SELECT @iLoopControl = 1

SELECT @iNextRowId = MIN(iRowId)

FROM CustomerTable

-- Make sure the table has data.

IF ISNULL(@iNextRowId,0) = 0

BEGIN

SELECT 'No data in found in table!'

RETURN

END

-- Retrieve the first row

SELECT @iCurrentRowId = iRowId,

@vchCustomerNmbr = vchCustomerNmbr,

@vchCustomerName = vchCustomerName

FROM CustomerTable

WHERE iRowId = @iNextRowId

-- start the main processing loop.

WHILE @iLoopControl = 1

BEGIN

-- This is where you perform your detailed row-by-row

-- processing.

-- Reset looping variables.

SELECT @iNextRowId = NULL

-- get the next iRowId

SELECT @iNextRowId = MIN(iRowId)

FROM CustomerTable

WHERE iRowId > @iCurrentRowId

-- did we get a valid next row id?

IF ISNULL(@iNextRowId,0) = 0

BEGIN

BREAK

END

-- get the next row.

SELECT @iCurrentRowId = iRowId,

@vchCustomerNmbr = vchCustomerNmbr,

@vchCustomerName = vchCustomerName

FROM CustomerTable

WHERE iRowId = @iNextRowId

END

RETURN

我们现在来看一下上面的游标，一般来说，为了性能的因素，我们的表上都有一个类似于RowID的列，此列可以被用来做循环，并得到相关的数据。一般来说此列是IDENTITY列，主键并有clustered索引。

但是很多情况下，我们的表中并不包括可以被用来循环的行ID，比如，可能在一个具有uniqueindentifier属性的列上创建了主键索引，这时候可以为这个表增加一个自增列并创建相应的索引，来实现此功能。

上面例子使用了MIN函数和”>” 来得到下一行我们需要的数据，当然我们也可以使用MAX函数然后使用”<”达到相同的功能。

***例子：***

SELECT @iNextRowId = MAX(iRowId)

FROM CustomerTable

WHERE iRowId < @iCurrentRowId

有一个比较重要的地方需要注意： 要在取得下一次要循环的行ID之前设置此ID为NULL，是因为当此循环实现所有的行循环后SELECT语句并不会为此ID设置为NULL，从而造成一个死循环。当循环变量为NULL后，意味着这个循环已经完成了它需要实现的功能，此时我们可能使用BREAK来退出WHILE循环，当然也有其它的途径来推出循环。

你可以在上面的存储过程中的如下注释处加上自己的基于行的操作。

-- This is where you perform your detailed row-by-row

-- processing.

可以看出，基于行的操作对性能非常有影响。举例来说，如果你有一个非常复杂的任务需要进行嵌套的循环，此时你会使用嵌套的游标，内层的游标根据外层游标的条件进行相应的操作，这时候如果我们使用游标来进行处理的话对服务器会有非常大的压力。

***例子：***

if exists (select \* from sysobjects where name = N'prcNestedLoopExample')

drop procedure prcNestedLoopExample

go

CREATE PROCEDURE prcNestedLoopExample

AS

/\*

\*\*put your comment here

\*/

SET NOCOUNT ON

-- declare all variables!

DECLARE @iReturnCode int,

@iNextCustRowId int,

@iCurrentCustRowId int,

@iCustLoopControl int,

@iNextProdRowId int,

@iCurrentProdRowId int,

@vchCustomerName nvarchar(255),

@chProductNumber nchar(30),

@vchProductName nvarchar(255)

-- Initialize variables!

SELECT @iCustLoopControl = 1

SELECT @iNextCustRowId = MIN(iCustId)

FROM Customer

-- Make sure the table has data.

IF ISNULL(@iNextCustRowId,0) = 0

BEGIN

SELECT 'No data in found in table!'

RETURN

END

-- Retrieve the first row

SELECT @iCurrentCustRowId = iCustId,

@vchCustomerName = vchCustomerName

FROM Customer

WHERE iCustId = @iNextCustRowId

-- Start the main processing loop.

WHILE @iCustLoopControl = 1

BEGIN

-- Begin the nested(inner) loop.

-- Get the first product id for current customer.

SELECT @iNextProdRowId = MIN(iProductId)

FROM CustomerProduct

WHERE iCustId = @iCurrentCustRowId

-- Make sure the product table has data for

-- current customer.

IF ISNULL(@iNextProdRowId,0) = 0

BEGIN

SELECT 'No products found for this customer.'

END

ELSE

BEGIN

-- retrieve the first full product row for

-- current customer.

SELECT @iCurrentProdRowId = iProductId,

@chProductNumber = chProductNumber,

@vchProductName = vchProductName

FROM CustomerProduct

WHERE iProductId = @iNextProdRowId

END

WHILE ISNULL(@iNextProdRowId,0) <> 0

BEGIN

-- Do the inner loop row-level processing here.

-- Reset the product next row id.

SELECT @iNextProdRowId = NULL

-- Get the next Product id for the current customer

SELECT @iNextProdRowId = MIN(iProductId)

FROM CustomerProduct

WHERE iCustId = @iCurrentCustRowId

AND iProductId > @iCurrentProdRowId

-- Get the next full product row for current customer.

SELECT @iCurrentProdRowId = iProductId,

@chProductNumber = chProductNumber,

@vchProductName = vchProductName

FROM CustomerProduct

WHERE iProductId = @iNextProdRowId

END

-- Reset inner loop variables.

SELECT @chProductNumber = NULL

SELECT @vchProductName = NULL

SELECT @iCurrentProdRowId = NULL

-- Reset outer looping variables.

SELECT @iNextCustRowId = NULL

-- Get the next iRowId.

SELECT @iNextCustRowId = MIN(iCustId)

FROM Customer

WHERE iCustId > @iCurrentCustRowId

-- Did we get a valid next row id?

IF ISNULL(@iNextCustRowId,0) = 0

BEGIN

BREAK

END

-- Get the next row.

SELECT @iCurrentCustRowId = iCustId,

@vchCustomerName = vchCustomerName

FROM Customer

WHERE iCustId = @iNextCustRowId

END

RETURN

 在上面的例子中，我们从一个表中进行循环，对于每个ID，我们从再从产品表中得到客户的相关产品信息。

SQL Server游标提供了一下非常有用并且强大的功能来实现基于行的操作，但是这种强大功能是依性能为代价的。上面我们列举了一种不使用游标来实现类似于游标。**这在很多场合可以有效提高性能。**

1. **使用存储过程**

当存储过程被第一次执行（没有使用WITH RECOMPILE选项），它的查询计划被编译并缓存在内存中，避免了下次再运行此存储过程使用要重新编译的问题，从而能够提高性能。

另外在存储过程中一定要使用”SET NOCOUNT ON”,如果不打开此选项，每次SQL SERVER运行此存储过程时都会返回给客户端每个语句影响的行数信息，如果客房端并不需要此信息，请打开此选项，这样可以减少网络流量及应用程序可能出现的问题。使用存储过程可以得到以下好处：

**存储过程可以提高性能**

当存储过程被创建,它要经过以下几步,第一步,它所包含的T-SQL语句将被分析和解析,并被存储.当第一次被执行时,它将被调出并被优化.SQLServer会根据statistics自动选择相应的优化策略.

此后查询计划将被存储在高速缓存中,以利于将来使用.由于不用被重新编译,所以可以大大提高效率.

**存储过程可以减少网络流量**

你可能使用T-SQL语句来对表执行插入操作.但是如果我们创建一个存储过程来进行这样操作的话,每次插入的时候只要传输存储过程名,参数和这些参数的数值,当这些操作非常频繁时我们将会发现使用存储过程可以减少额外的网络传输.这在我们使用Internet时进行传输时非常有用.

比较以下两个语句:

INSERT INTO EmployeeTerritories (EmployeeID, TerritoryID)

VALUES (3,12345)

Ins\_EmployeeTerritories @empId=3,@terrId=12345

第一个语句有74个字符,第二个有46个字符,相比而言网络传输量减少了37.84%,如果我们这个插入语句中包括有更多要插入数据的列,并且每天被执行10,000次,将会增加280K左右的带宽,

**存储过程可以增加安全性**

如可以有效减少注入式攻击的风险：

SqlCommand command = new SqlCommand("proc\_IsUserValid", connection);

command.CommandType = CommandType.StoredProcedure;

command.Parameters.Add("@username", SqlDbType.VarChar).Value = username;

command.Parameters.Add("@password", SqlDbType.VarChar).Value = password;

command.Parameters.Add("@return", SqlDbType.Int).Direction =

ParameterDirection.ReturnValue;

int count = (int)command.ExecuteScalar();

所以我们应该尽量在我们的应用系统中使用存储过程.

1. **不要包括不做任何事情的代码.**

大家都可能都认为这是一个显而易见的问题，但是在我们的代码中有一些为了前台应用程序的拼接SQL的方便，写下了如下的代码：

SELECT column\_name FROM table\_name

WHERE 1 = 1 and …….

当这个查询中的条件1=1运行时，并不做任何事情，并且使用SQL Server 的资源，所以要尽量少使用这样的语句。

1. **sp\_?**

强烈建议所有的用户自定义存储过程都不要使用sp\_作为前缀，因为这个前缀标识的是系统存储过程，当你执行一个sp\_前缀的存储过程时SQL Server会按以下的次序执行：

* 1. 查找master数据库的存储过程
  2. 查找当前用户的user下是否有这个存储过程
  3. 查找dbo是否拥有这个存储过程

此外还有基于表意性的考虑，因为当我们看到sp\_XXXX时，我们总会想到是系统存储过程，增加后期的维护的成本。

1. **CHECKSUM索引**

下面我们通过例子来看一下Checksum是如何工作的，比如有一个存储客户信息的数据库，我们一个客户的登录名是”daviddong@newegg.com”,我们可以看到此列是非常宽的，也就是说如果我们在此列上创建索引的话我们的索引将占用非常多的空间，从而影响性能，为了解决此问题，我们可能加入一个Checksum列，此Checksum列存储客户登录名字的Checksum值，然后我们为此哈希列创建索引，这样我们就能够很大的提高性能。

Checksum能够实现此功能的原因之一是对于相同的输入，它返回相同的值，也就是说它的结果和时间，环境因素无关。当我们使用它也不是使用登录名字为创建索引后，可能有人会问，我们的索引所占用的空间是少了，但是我Search的时候是希望按照登录的名字来查找的啊，因为谁能够记住近四十亿个不同的Checksum值啊？

下面我们来看一下下面的T-SQL:

SELECT Customernumber, CustomerName, LoginName

FROM Customer

WHERE LoginName = 'DavidDong@newegg.com'

AND checksum\_ LoginName = CHECKSUM('DavidDong@newegg.com')

如上例，因为我又加上了一个And操作，从而在比较的时候要进行两次比较，是这样的，我们进行了两次比较，这只因为我们不能够确保Checksum值的唯一性（不同的歌曲名可能会产生同一个Checksum值）。看上去我们的查询性能可能会更差，现在我们来仔细看一下上面的代码。由于Checsum\_title列具有高选择性（最小的重复值），查询优化器会首先选择此索引，然后根据此索引导航到相应的行再进行Title的比较，这样性能其实会有很多提高的。这个在开发时需要很多额外的人力工作量，但是一般情况下，我们可以从checksum值上得到比较好的性能提升。

1. **索引提示（Index hint）**

如果我们在在WHERE子句中使用IN选项或者OR选项来进行操作时，如果发现SQL Server使用表扫描而不是使用索引（有时候是这几列的组合索引时）来查找数据时，我们可以考虑使用索引提示来强制SQL Server使用此索引。

比如：

SELECT \* FROM tblTaskProcesses

WHERE nextprocess = 1 AND processid IN (8,32,45)

如果加上索引提示，可以被重写为下面的语句:

SELECT \* FROM tblTaskProcesses (INDEX = IX\_ProcessID)

WHERE nextprocess = 1 AND processid IN (8,32,45)

1. **事务和死锁**
2. **Transaction**

事务作为一个逻辑单元，必须具备四个属性：自动性、一致性、独立性和持久性。事务机制保证一组数据的修改要么全部执行，要么全不执行。SQL Server使用事务保证数据的一致性和确保在系统失败时的可恢复性。

事务是一个可以恢复的单元的工作，由一条或者多条Transact-SQL语句组成，可以影响到表中的一行或者多行数据。事务打开以后，直到事务成功完成之后提交为止，或者到事务执行失败全部取消或者滚回去为止，下面是事务的工作原理：



图1   事务的工作原理图

在使用事务时，原则上应该使事务尽可能得短并且要避免事务嵌套。

事务应该尽可能得短，这是因为比较长的事务增加了事务占用数据的时间，使其它必须等待访问该事务锁定数据的事务，延长了等待访问数据的时间。为了最小化时间，在使用一些Transact-SQL语句时，一定避免在事务中于用户的交互。例如，在开始事务之前，一定要知道需要用户交互式操作才能得到的信息。这样，在事务的进行过程中，就可以避免进行一些耗费时间的交互式操作，缩短事务进程的时间。

在一个用户定义的事务中，应该尽可能地使用一些数据操纵语言，例如INSERT、UPDATE和DELETE语句，因为这些语句主要是操纵数据库中的数据。而对于一些数据定义语言，应该尽可能地少用或者不用，因为这些数据定义语言的操作既占用比较长的时间，又占用比较多的资源，并且这些数据定义语言的操作通常不涉及到数据，所以应该在事务中尽可能地少用或者不用这些操作。另外，在使用数据操纵语言时，要注意，一定要在这些语句中使用条件判断语句，使得这些数据操纵语言涉及到尽可能少的记录，从而缩短事务的处理时间。

在嵌套事务时，也要注意一些问题。因为使用嵌套事务，除了把事务搞得更加复杂之外，并没有这么明显的好处。因此，不建议使用嵌套事务。

1. **Complex transaction**

如果创建了一个复杂的事务，这个事务包括几个部分，其中一个部分和其它部分相比有较大ROLL BACK的可能性，这样事务在回滚的时候就会消耗比较小的资源。从而提高资源的利用率。

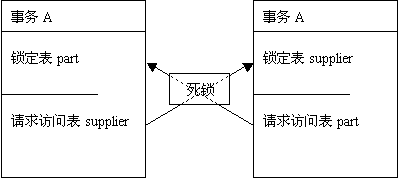
另外，建议不要使用嵌套式事务。

1. **Deadlock**

死锁是很重要的一个地方。在事务和锁的使用过程中，死锁是一个不可避免的现象。

**死锁产生的情况：**

第一种情况是，当两个事务分别锁定了两个单独的对象，这时每一个事务都要求在另外一个事务锁定的对象上获得一个锁，因此每一个事务都必须等待另外一个事务释放占有的锁，这时，就发生了死锁。这种死锁是最典型的死锁形式。在同一时间内有两个事务A和B，事务A有两个操作：锁定表part和请求访问表supplier；事务B也有两个操作：锁定表supplier和请求访问表part。结果，事务A和事务B之间发生了死锁，其示意进程如下图所示。



死锁的第二种情况是，当在一个数据库中时，有若干个长时间运行的事务执行并行的操作，当查询分析器处理一种非常复杂的查询例如连接查询时，那么由于不能控制处理的顺序，有可能发生死锁现象。

**死锁的信息的收集与分析与避免方法**

并发用户数多、数据量大的情况下，如果开发时又没有特别的关注Performance与Deadlock的问题，SQL Server很容易产生deadlock，Deadlock发生时会产生类似

"Transaction (Process ID %d) was deadlocked on {%Z} resources with another process and has been chosen as the deadlock victim. Rerun the transaction."

的报错信息。

Deadlock一般发生在交互操作相当频繁的Table、Proc上，很难说出绝对的解决问题，只是将发生的机会降到最低。Update/Delete语句中，如我们没有指定相关的表提示而让SQL Server自行控制，SQL Server会试图寻找最合适的Lock Type。先根据WHERE/JOIN等条件以及当前可用的Index进行扫描，定位要操作的数据位置或范围，然后加锁，进行后续的操作。 例如，如果根据Index找到了要更新的某记录行，则可能对该行使用ROWLOCK；如果根据Index定位到了几个数据页，则可能对这些数据页使用PAGLOCK；如果没有索引或索引与条件不符合，则可能使用TABLOCK，尽管更新的可能只是一条记录，SQL Server也要执行一次Full table scan，并且这期间使用了TABLOCK（这种情况下，不仅Update/Delete语句本身或者是并发的语句之间存在严重的效率问题，如果Isolation为ReadCommitted等，Select语句也需要排队等待，如果这个Table的存取相当频繁，那涉及到这个Table的操作就都会变得相当慢）。Update/Delete的条件一定要充分的使用Index。并且就算你在表提示上加上rowlock,SQL Server也会在必要的升级锁。一个强制SQL Server不进行表升级的方法就是找开一个特定的跟踪选项。

**死锁信息收集**

Deadlock经常发生，有以下几种方法收集死锁的停息从而解决死锁问题：

第一种方法: 打开Deadlock trace选项。

DBCC TRACEON(1204,3605,-1)

GO

DBCC TRACESTATUS(-1)

GO

其中1204作用是Deadlock发生时收集相关的信息；3605作用是将收集的信息写入SQL Server的Error Log中。Trace选项打开后，在SQL Server Enterprise Manager的Management->SQL Server Logs下可以看到相关的Log。

第二种方法: 使用Profiler捕获详细信息。

有时候Deadlock发生时，使用trace选项从Error Log中并不能发现；另外Profiler记录了更详细的信息，包括Deadlock发生时所执行的Transact-SQL。方便起见，使用Profiler的Trace时，将默认的Events都去掉，只选择Locks下面的Deadlock。  Profiler Trace中的Deadlock对应于1204 Trace选项

**死锁信息的分析**

在SQL Server Enterprise Manager的Management->SQL Server Logs中可以查看到Deadlock发生时收集到的信息。如果打开了1205 Trace选项，则记录的Log会很多，可以用在Log中查找Deadlock的发生点。

**如何避免产生死锁**

上面也只是出现了deadlock后采取的措施，其它我们在应用程序开发时就应该考虑到这个问题，为了最小化死锁的发生，请参照下面的建议：

* 依次访问数据库对象
* 事务中不能够允许输入
* 保持事务尽可能短
* 如果可能，使用尽可能低的事务隔离级别

1. **表操作**
   1. **Avoid Table scan**

Table Scan会导致SQL Server使用过多的I/O，浪费SQL Server 的资源。也会增加网络带宽，影响SQL Server性能。如果一个表很大的话，可能会产生会在扫描的时候锁死表就阻塞其它用户的访问。

此外，表扫描也会清空缓存中的数据，并把很多无用的数据填充进去，从而使SQL Server不能够重用缓存中的数据，增加了I/O并影响性能。

* 1. **SELECT INTO**

使用SELECT INTO子句时，这个语句会锁定相关的系统表，同时阻塞其它用户对表的访问。如果一定要使用SELECT INTO语句，尽量在SQL Server比较空闲是运行，并尽可能减少插入行的数目。

* 1. **Update**

由于UPDATE要占用很多资源，所以书写UPDATE语句时，为了减少UPDATE语句占用的资源，请参照以下的建议：

* + - 如果你要更改的列上具有Unique索引，请试图每次只更新一行。
    - 尽量不要更新主键列
    - 当更新一个VARCHAR列时，尽量更新为相同的长度，或者比原来长度要小（否则会产生分页，从而会影响性能）
    - 尽量不在要频繁更新的表上创建FOR UPDATE的触发器
    - 大批量更新被复制的表时，请尽可能减少每次更新影响的行数。

很多情况下进行数据更新时没有选择，上面只是一些在更新的时候要考虑的建议。

* 1. **大批量更新，删除，插入数据**

大批量更新，删除和插入数据时，由于要考虑到不阻塞其它用户的活动，复制的活动量的原因，所以要控制每次操作影响的行数。

一方面是提高并发性：在不打开SQL Server特定TRACE的情况下，就算我们在进行这些操作时使用了WITH(Rowlock)，SQL Server也会在必要的时候进行锁升级，这样就会阻塞其它用户的访问，从而降低并发。

另一方面是复制：如果在被复制表的大数据量数据库操作（更新，删除和插入数据）时，在SQL Server 2000中会出面一些问题(可能会导致服务器资源耗尽，在SQL Server 2005中这方面功能有了增强)。

***例子：***

如果在发布服务器中创建了下面的存储过程：

CREATE PROC give\_raise AS

UPDATE EMPLOYEES SET salary = salary \* 1.10

此存储过程完成让公司内部10,000个雇员薪水增长10%的任务，但在发布服务器上执行此存储时，它对每个雇员的薪水进行更新，那么此更新将向订阅服务器发送一个非常大的，多阶段事务，执行10,000次update，如下：

BEGIN TRAN

UPDATE EMPLOYEES SET salary = salary \* 1.10 WHERE PK = 'emp 1'

UPDATE EMPLOYEES SET salary = salary \* 1.10 WHERE PK = 'emp 2'

所以在服务器上进行大数据量的操作，可能会导致服务器资源耗尽。

为了解决这些问题，建议使用循环更新的方法来进行:

SET ROWCOUNT 200                       --> THIS ALLOWS ONLY 10,000 ROWS TO BE DELETED

GO

DECLARE @NO\_ROWS\_LEFT INT

SELECT @NO\_ROWS\_LEFT=2

WHILE @NO\_ROWS\_LEFT <> 0

BEGIN

    DELETE FROM  imk.dbo.VerifyVisaResult

    WHERE  Indate<’xxxx’

    SELECT @NO\_ROWS\_LEFT=@@ROWCOUNT

    PRINT @NO\_ROWS\_LEFT

END

* 1. **TempTable & Derived Tables**

临时表可能会影响性能。主要因为在使用临时表时有额外的操作，为了有最大的性能提高，要让额外的操作越少越好。例如，使用SELECT语句，SQL Server从硬盘上读数据并且返回数据，但是临时表需要系统做一些其它额外的操作。

比如，在T-SQL中使用临时表时，一般如下使用：

* 创建临时表（写）
* 向新创建的临时表中插入数据（读->写）
* 查询临时表的数据，一般是和其它物理表进行JOIN（读）
* 删除临时表（写）

由上面的几个步骤可以看到要有很多额外的操作。可以使用Derived table来减少上面的步骤从而提高性能。使用Derived table时，所有的数据库活动都发生在内存中而不是临时表时的内存和磁盘，这就减少了系统的活动，减少了磁盘I/O,加快了查询。

当使用Drived table时，只是简单的从内存中读数据，综上所述，我们知道使用Derived table可以减少I/O,从而提升性能。使用Derived Table其实就是在SELECT语句中使用基本的SELECT语句。

***例子：***

什么是Derived Table及它的使用，下面一个最简单的查询：

USE northwind

GO

SELECT \* FROM dbo.Categories

如果使用了Dervied table，如下：

USE northwind

GO

SELECT \* FROM (SELECT \* FROM dbo.Categories) as DerivedTable

本质上就是不直接从基本表中读取数据而已。事实上，Derived table就是用另一个SELECT语句查询而得到的一个结果集，然后用小括号括起来再加个别名，此后我们就可以像使用正常表一样来使用Derived table了。

上面的两个例子查询返回相同的结果集，这个例子中只简单的示例怎么使用Derived table，如果要确实在程序里实现上面查询时，肯定是使用第一个简单的查询语句，当查询非常复杂时，考虑使用Derived table来提升性能。

这个例子里，我们使用Northwind数据库（Northwind数据库SQL Server 2000的示例数据库）。现在需要得到在每个Categories中的product列表和product个数，就是希望得到如下结果集：

|  |  |  |
| --- | --- | --- |
| **Category Name**> | **Product Name**> | **Category Count**> |
| Beverages | Outback Lager | 2 |
| Beverages | Chang | 2 |
| Condiments | Aniseed Syrup | 3 |
| Condiments | Cajun Seasoning | 3 |
| Condiments | Gumbo Mix | 3 |

第一个查询是使用临时表来实现：

-- CREATE OUR TEMPORARY TABLE

CREATE TABLE #Temp\_Example (

[CategoryID] INT NOT NULL,

[Category\_Count] INT NOT NULL

)

-- INSERT THE VALUES WE LATER NEED INTO THE TEMP TABLE

INSERT INTO #Temp\_Example (CategoryID, Category\_Count)

SELECT C.CategoryID, COUNT(\*) AS Category\_Count

FROM Categories C

INNER JOIN Products P ON C.CategoryID = P.CategoryID

GROUP BY C.CategoryID, C.CATEGORYNAME

-- JOIN ON THE TEMP TABLE TO GET OUR VALUES

SELECT C.CategoryID, C.CategoryName, P.ProductName, P.UnitPrice, #Temp\_Example.Category\_Count

FROM Categories C

INNER JOIN Products P ON C.CategoryID = P.CategoryID

INNER JOIN #Temp\_Example ON C.CategoryID = #Temp\_Example.CategoryID

ORDER BY C.CategoryName

-- DROP TEMPORARY TABLE

DROP TABLE #Temp\_Example

GO

下面我们来使用Derived table而不是使用临时表来实现：

-- NOTE HOW WE SIMPLY JOIN ON THE TABLE CREATED IN MEMORY BASED ON THE CATEGORY ID

SELECT C.CategoryID, C.CategoryName, P.ProductName, P.UnitPrice, CT.Category\_Count

FROM Categories C

INNER JOIN Products P ON C.CategoryID = P.CategoryID

INNER JOIN (

SELECT C.CategoryID, COUNT(\*) AS Category\_Count

FROM Categories C

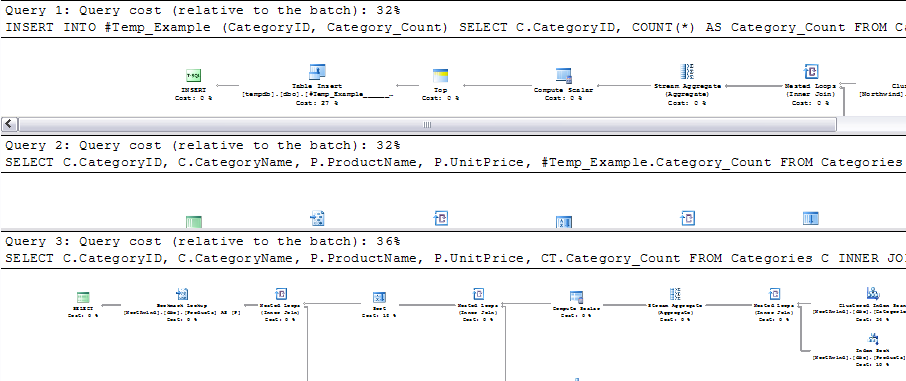
INNER JOIN Products P ON C.CategoryID = P.CategoryID

GROUP BY C.CategoryID, C.CategoryName

)CT ON C.CategoryID = CT.CategoryID

ORDER BY C.CategoryName

参照下面的执行计划，可见第一个查询(使用了临时表)占了总查询成本的64%，而第二个使用Derived Table的查询占总成本的36%，可见第二种写法可以显著的提高查询性能。



提高性能主要是由于以下原因：

* 并不直接向临时表进行显示的插入操作，减少I/O操作
* 所有的JOIN操作现在都是在内存中进行，而不是在临时表中

现在我们可以看到使用Derived Table在解决复杂性能问题的时候可能提高性能。

* 1. **有效的使用JOIN**

表的JOIN操作一般都是导致性能问题的罪魁祸首，特别是当JOIN中包括两个以上表或者表中数据量非常大的时候。但JOIN在OLTP数据库中是肯定要存在的，也是必需要存在的。

* 如果我们要经常对两个或者多个表进行JOIN操作，首先JOIN列一定要有一个适当的索引。如果用于JOIN的列非常宽或者其它原因，不能够创建索引，请考虑使用代理键.
* 用于JOIN的列一定要是同一种数据类型，并且最好是数值型的，因为数据型在进行比较的时候比字条型要高效。
* 避免JOIN的列上只有比例很少的数据有唯一值，如果JOIN列只有比例很少的唯一值，SQL Server优化器会进行表扫描而不是使用在JOIN列上的索引进行相应的JOIN操作，所以大大降低性能，为了提高性能，我们最好能够在JOIN时使用具有唯一值的列上进行（只是建议，因为现实中不可能用于JOIN的列都是唯一键）
* 如果被JOIN的表中包括4个或者更多的表的话，请考虑在设计进行反范式化设计，从而达到更好的性能。一般情况下可能在一些表上增加一列来减少JOIN表的数量。
  1. **父子表**

我们的数据库中一般都有很多父子型的表，比如订单表和订单明细表，在这些表有经常要执行一些操作，一个常见的操作就是查找在父表中存在但是在子表中不存在的记录。一般情况下我们可能通过三种途径来解决这个问题：

--使用NOT EXISTS

SELECT a.hdr\_key

FROM hdr\_tbl a

WHERE NOT EXISTS (SELECT \* FROM dtl\_tbl b WHERE a.hdr\_key = b.hdr\_key)

--使用LEFT JOIN

SELECT a.hdr\_key

FROM hdr\_tbl a

LEFT JOIN dtl\_tbl b ON a.hdr\_key = b.hdr\_key

WHERE b.hdr\_key IS NULL

--使用NOT IN

SELECT hdr\_key

FROM hdr\_tbl

WHERE hdr\_key NOT IN (SELECT hdr\_key FROM dtl\_tbl)

在上面的三种方法中，都能够返回正确的结果，但上面的三种方法中Not EXISTS最好，而 NOT IN最差。不过使用时，最好对上面的每个语句都进行一下测试，因为有不同的索引，不同的数据量等

* 1. **关于统计信息（Statistics）**

如果我们的数据被设置”Auto Create Statistics"后,SQL Server 查询优化器在运行查询时，会为当前不存在状态信息的列增加状态信息以提升性能。当然这是一件好事，因为统计信息提供SQL Server 查询优化器相应的信息，以使SQL Server在执行查询的时候可以自动选择一个比较优化的执行计划。

其实SQL Server为列自动增加统计信息的做法是有很多好处的，因为它从一个侧面出反映我们应该考虑是否为此列上创建相关的索引。如果查询优化器认为某列的统计信息有用，但是我们最好可以检查一下是否能够在此列上创建索引，在确保能够有限提高性能（存在统计信息并不一定说明我们要在此列上创建索引）。我们在查询分析器中运行下面的语句就可以得到查询优化器自动为哪些列的统计信息：

USE PUBS;

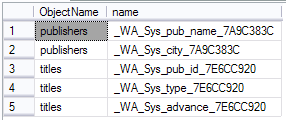
GO

SELECT object\_name(id) ObjectName,name

FROM SYSINDEXES

WHERE (NAME LIKE '%\_WA\_SYS%')

结果如下图,我们可以看到现在SQL Server在Titile表的pub\_id,type和advance列上创建了统计信息,此时我们就可以考虑是否可以在这些列上创建索引来提高性能：



但是当SQL Server数据库具在非常大的负载的时候，此选项会带来数据装载时的性能消耗，为了解决此问题，我们可以定时在数据库压力比较小的时候来更新数据库状态信息。使用 UPDATE STATISTICS 或者 sp\_updatestats。不过关闭此选项后可能会减轻一些数据库服务器的压力，但是也会造成我们的一些查询无法得到最佳的优化，这也会造成服务器压力的增大。所以一般情况下，我们很多服务器并不是处于一个非常大的压力之下，还是建议打开此选项.

* 1. **为列选择合适数据类型**

选择合适的数据类型可以大大提高数据库查询，插入，更新和删除的性能。下面是我们基于性能考虑在选择数据类型时的建议

* + - 永远选择占用空间最少的数据类型来存储数据：如果我们只是用来存储1到100的100个数字，这时TINYINT和INT相比会是更合适的数据类型。这个原理对于CHAR和VARCHAR依然适用。选择合适的数据类型可以减少数据的存储空间以及索引占用的空间，从而减少I/O的数量，提高网络传输时间及减少时延。
    - 如果字符数据的长度会有非常大的改变话,使用VARCHAR来替代CHAR，此时尽管VARCHAR数据类型一般来说会有降低，但是考虑到所占用空间的减少及相关I/O的减少，VARCHAR会有性能的提升。
    - 除非要存储16位的UNICODE 数据，否则不要使用NVARCHAR或者NCHAR数据类型。
    - 如果我们需要存储大量字符数据，但是会小于是8000个字符的话，使用VARCHAR数据类型来替代TEXT数据类型，因为TEXT会有额外的开销从而降低性能和可维护性。
    - 如果我们有一列用来存放数字的数据，使用数据类型，如INT来替代VARCHAR 或者CHAR类型，因为INT来存储数字可以有效节省空间（减少I/O），并提高查找，比较和JOIN的性能。
    - Char Vs Varchar:一般来说Varchar在插入和查询时性能要稍好一些，但是在更新时CHAR性能较好。因为空间已经分好，不用做额外的分页。

1. **参考资料**