**SQL索引一步到位**

 　　SQL索引在数据库优化中占有一个非常大的比例， 一个好的索引的设计，可以让你的效率提高几十甚至几百倍，在这里将带你一步步揭开他的神秘面纱。

**1.1 什么是索引？**

　　SQL索引有两种，聚集索引和非聚集索引，索引主要目的是提高了SQL Server系统的性能，加快数据的查询速度与减少系统的响应时间

下面举两个简单的例子：

图书馆的例子：一个图书馆那么多书，怎么管理呢？建立一个字母开头的目录，例如：a开头的书，在第一排，b开头的在第二排，这样在找什么书就好说了，这个就是一个聚集索引，可是很多人借书找某某作者的，不知道书名怎么办？图书管理员在写一个目录，某某作者的书分别在第几排，第几排，这就是一个非聚集索引

字典的例子：字典前面的目录，可以按照拼音和部首去查询，我们想查询一个字，只需要根据拼音或者部首去查询，就可以快速的定位到这个汉字了，这个就是索引的好处，拼音查询法就是聚集索引，部首查询就是一个非聚集索引.

    看了上面的例子，下面的一句话大家就很容易理解了：聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续。就像字段，聚集索引是连续的，a后面肯定是b，非聚集索引就不连续了，就像图书馆的某个作者的书，有可能在第1个货架上和第10个货架上。还有一个小知识点就是：聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个。

**1.2 索引的存储机制**

  　　首先，无索引的表，查询时，是按照顺序存续的方法扫描每个记录来查找符合条件的记录，这样效率十分低下,举个例子，如果我们将字典的汉字随即打乱，没有前面的按照拼音或者部首查询，那么我们想找一个字，按照顺序的方式去一页页的找，这样效率有多底，大家可以想象。

       聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致，其实理解起来非常简单，还是举字典的例子：如果按照拼音查询，那么都是从a-z的，是具有连续性的，a后面就是b，b后面就是c， 聚集索引就是这样的，他是和表的物理排列顺序是一样的，例如有id为聚集索引，那么1后面肯定是2,2后面肯定是3，所以说这样的搜索顺序的就是聚集索引。非聚集索引就和按照部首查询是一样是，可能按照偏房查询的时候，根据偏旁‘弓’字旁，索引出两个汉字，张和弘，但是这两个其实一个在100页，一个在1000页，（这里只是举个例子），他们的索引顺序和数据库表的排列顺序是不一样的，这个样的就是非聚集索引。

      原理明白了，那他们是怎么存储的呢？在这里简单的说一下，聚集索引就是在数据库被开辟一个物理空间存放他的排列的值，例如1-100，所以当插入数据时，他会重新排列整个整个物理空间，而非聚集索引其实可以看作是一个含有聚集索引的表，他只仅包含原表中非聚集索引的列和指向实际物理表的指针。他只记录一个指针，其实就有点和堆栈差不多的感觉了

**1.3 什么情况下设置索引**

|  |  |  |
| --- | --- | --- |
| 动作描述 | 使用聚集索引 | 使用非聚集索引 |
| 外键列 | 应 | 应 |
| 主键列 | 应 | 应 |
| 列经常被分组排序(order by) | 应 | 应 |
| 返回某范围内的数据 | 应 | 不应 |
| 小数目的不同值 | 应 | 不应 |
| 大数目的不同值 | 不应 | 应 |
| 频繁更新的列 | 不应 | 应 |
| 频繁修改索引列 | 不应 | 应 |
| 一个或极少不同值 | 不应 | 不应 |

建立索引的原则：

1) 定义主键的数据列一定要建立索引。

2) 定义有外键的数据列一定要建立索引。

3) 对于经常查询的数据列最好建立索引。

4) 对于需要在指定范围内的快速或频繁查询的数据列;

5) 经常用在WHERE子句中的数据列。

6) 经常出现在关键字order by、group by、distinct后面的字段，建立索引。如果建立的是复合索引，索引的字段顺序要和这些关键字后面的字段顺序一致，否则索引不会被使用。

7) 对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。

8) 对于定义为text、image和bit的数据类型的列不要建立索引。

9) 对于经常存取的列避免建立索引

9) 限制表上的索引数目。对一个存在大量更新操作的表，所建索引的数目一般不要超过3个，最多不要超过5个。索引虽说提高了访问速度，但太多索引会影响数据的更新操作。

10) 对复合索引，按照字段在查询条件中出现的频度建立索引。在复合索引中，记录首先按照第一个字段排序。对于在第一个字段上取值相同的记录，系统再按照第二个字段的取值排序，以此类推。因此只有复合索引的第一个字段出现在查询条件中，该索引才可能被使用,因此将应用频度高的字段，放置在复合索引的前面，会使系统最大可能地使用此索引，发挥索引的作用。

**1.4 如何创建索引**

1.41 创建索引的语法：

CREATE [UNIQUE][CLUSTERED | NONCLUSTERED]  INDEX  index\_name

ON {table\_name | view\_name} [WITH [index\_property [,....n]]

说明：

UNIQUE: 建立唯一索引。

CLUSTERED: 建立聚集索引。

NONCLUSTERED: 建立非聚集索引。

Index\_property: 索引属性。

 UNIQUE索引既可以采用聚集索引结构，也可以采用非聚集索引的结构，如果不指明采用的索引结构，则SQL Server系统默认为采用非聚集索引结构。

1.42 删除索引语法：

DROP INDEX table\_name.index\_name[,table\_name.index\_name]

说明：table\_name: 索引所在的表名称。

index\_name : 要删除的索引名称。

1.43 显示索引信息：

使用系统存储过程：sp\_helpindex 查看指定表的索引信息。

执行代码如下：

Exec sp\_helpindex book1;

**1.5 索引使用次数、索引效率、占用CPU检测、索引缺失**

　　当我们明白了什么是索引，什么时间创建索引以后，我们就会想，我们创建的索引到底效率执行的怎么样？好不好？我们创建的对不对？

　　首先我们来认识一下DMV，DMV (dynamic management view)动态管理视图和函数返回特定于实现的内部状态数据。推出SQL Server 2005时，微软介绍了许多被称为dmvs的系统视图，让您可以探测SQL Server 的健康状况，诊断问题，或查看SQL Server实例的运行信息。统计数据是在SQL Server运行的时候开始收集的，并且在SQL Server每次启动的时候，统计数据将会被重置。当你删除或者重新创建其组件时，某些dmv的统计数据也可以被重置，例如存储过程和表，而其它的dmv信息在运行dbcc命令时也可以被重置。

　　当你使用一个dmv时，你需要紧记SQL Server收集这些信息有多长时间了，以确定这些从dmv返回的数据到底有多少可用性。如果SQL Server只运行了很短的一段时间，你可能不想去使用一些dmv统计数据，因为他们并不是一个能够代表SQL Server实例可能遇到的真实工作负载的样本。另一方面，SQL Server只能维持一定量的信息，有些信息在进行SQL Server性能管理活动的时候可能丢失，所以如果SQL Server已经运行了相当长的一段时间，一些统计数据就有可能已被覆盖。

　　因此，任何时候你使用dmv，当你查看从SQL Server 2005的dmvs返回的相关资料时，请务必将以上的观点装在脑海中。只有当你确信从dmvs获得的信息是准确和完整的，你才能变更数据库或者应用程序代码。

下面就看一下dmv到底能带给我们那些好的功能呢？

1.51 ：索引使用次数

我们下看一下下面两种查询方式返回的结果（这两种查询的查询用途一致）

①----

declare @dbid int

select @dbid = db\_id()

select objectname=object\_name(s.object\_id), s.object\_id, indexname=i.name, i.index\_id, user\_seeks, user\_scans, user\_lookups, user\_updates

from sys.dm\_db\_index\_usage\_stats s, sys.indexes i

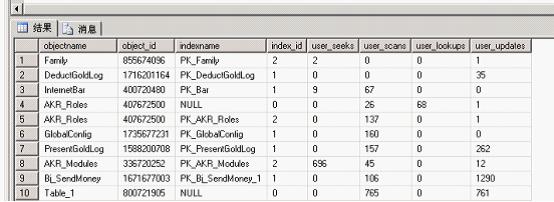
where database\_id = @dbid and objectproperty(s.object\_id,'IsUserTable') = 1

and i.object\_id = s.object\_id

and i.index\_id = s.index\_id

order by (user\_seeks + user\_scans + user\_lookups + user\_updates) asc

返回查询结果



②：使用多的索引排在前面

SELECT  objects.name ,

        databases.name ,

        indexes.name ,

        user\_seeks ,

        user\_scans ,

        user\_lookups ,

        partition\_stats.row\_count

FROM    sys.dm\_db\_index\_usage\_stats stats

        LEFT JOIN sys.objects objects ON stats.object\_id = objects.object\_id

        LEFT JOIN sys.databases databases ON databases.database\_id = stats.database\_id

        LEFT JOIN sys.indexes indexes ON indexes.index\_id = stats.index\_id

                                         AND stats.object\_id = indexes.object\_id

        LEFT  JOIN sys.dm\_db\_partition\_stats partition\_stats ON stats.object\_id = partition\_stats.object\_id

                                                              AND indexes.index\_id = partition\_stats.index\_id

WHERE   1 = 1

--AND databases.database\_id = 7

        AND objects.name IS NOT NULL

        AND indexes.name IS NOT NULL

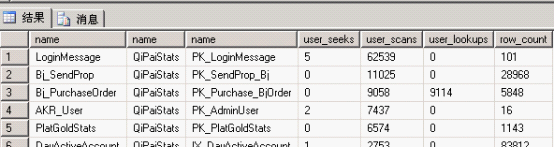
        AND user\_scans>0

ORDER BY user\_scans DESC ,

        stats.object\_id ,

        indexes.index\_id

返回查询结果



user\_seeks : 通过用户查询执行的搜索次数。   
 个人理解： 此统计索引搜索的次数

user\_scans: 通过用户查询执行的扫描次数。   
  个人理解：此统计表扫描的次数，无索引配合  
user\_lookups: 通过用户查询执行的查找次数。   
 个人理解：用户通过索引查找，在使用RID或聚集索引查找数据的次数，对于堆表或聚集表数据而言和索引配合使用次数  
user\_updates:  通过用户查询执行的更新次数。   
  个人理解：索引或表的更新次数

我们可以清晰的看到，那些索引用的多，那些索引没用过，大家可以根据查询出来的东西去分析自己的数据索引和表

1.52 ：索引提高了多少性能

新建了索引到底增加了多少数据的效率呢？到底提高了多少性能呢？运行如下SQL可以返回连接缺失索引动态管理视图，发现最有用的索引和创建索引的方法：

SELECT

avg\_user\_impact AS average\_improvement\_percentage,

avg\_total\_user\_cost AS average\_cost\_of\_query\_without\_missing\_index,

'CREATE INDEX ix\_' + [statement] +

ISNULL(equality\_columns, '\_') +

ISNULL(inequality\_columns, '\_') + ' ON ' + [statement] +

' (' + ISNULL(equality\_columns, ' ') +

ISNULL(inequality\_columns, ' ') + ')' +

ISNULL(' INCLUDE (' + included\_columns + ')', '')

AS create\_missing\_index\_command

FROM sys.dm\_db\_missing\_index\_details a INNER JOIN

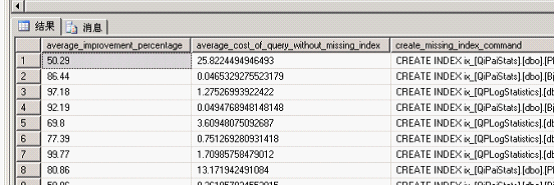
sys.dm\_db\_missing\_index\_groups b ON a.index\_handle = b.index\_handle

INNER JOIN sys.dm\_db\_missing\_index\_group\_stats c ON

b.index\_group\_handle = c.group\_handle

WHERE avg\_user\_impact > = 40

返回结果



虽然用户能够修改性能提高的百分比，但以上查询返回所有能够将性能提高40%或更高的索引。你可以清晰的看到每个索引提高的性能和效率了

1.53 ：最占用CPU、执行时间最长命令

这个和索引无关，但是还是在这里提出来，因为他也属于DMV带给我们的功能吗，他可以让你轻松查询出，那些sql语句占用你的cpu最高

SELECT TOP 100 execution\_count,

           total\_logical\_reads /execution\_count AS [Avg Logical Reads],

           total\_elapsed\_time /execution\_count AS [Avg Elapsed Time],

                db\_name(st.dbid) as [database name],

           object\_name(st.dbid) as [object name],

           object\_name(st.objectid) as [object name 1],

           SUBSTRING(st.text, (qs.statement\_start\_offset / 2) + 1,

           ((CASE statement\_end\_offset WHEN - 1 THEN DATALENGTH(st.text) ELSE qs.statement\_end\_offset END - qs.statement\_start\_offset)

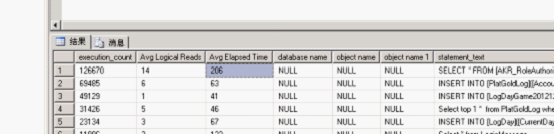
             / 2) + 1) AS statement\_text

  FROM sys.dm\_exec\_query\_stats AS qs CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) AS st

 WHERE execution\_count > 100

 ORDER BY 1 DESC;

返回结果：



执行时间最长的命令

SELECT TOP 10 COALESCE(DB\_NAME(st.dbid),

DB\_NAME(CAST(pa.value as int))+'\*',

'Resource') AS DBNAME,

SUBSTRING(text,

-- starting value for substring

        CASE WHEN statement\_start\_offset = 0

OR statement\_start\_offset IS NULL

THEN 1

ELSE statement\_start\_offset/2 + 1 END,

-- ending value for substring

        CASE WHEN statement\_end\_offset = 0

OR statement\_end\_offset = -1

OR statement\_end\_offset IS NULL

THEN LEN(text)

ELSE statement\_end\_offset/2 END -

CASE WHEN statement\_start\_offset = 0

OR statement\_start\_offset IS NULL

THEN 1

ELSE statement\_start\_offset/2  END + 1

)  AS TSQL,

total\_logical\_reads/execution\_count AS AVG\_LOGICAL\_READS

FROM sys.dm\_exec\_query\_stats

CROSS APPLY sys.dm\_exec\_sql\_text(sql\_handle) st

OUTER APPLY sys.dm\_exec\_plan\_attributes(plan\_handle) pa

WHERE attribute = 'dbid'

ORDER BY AVG\_LOGICAL\_READS DESC ;



看到了吗？直接可以定位到你的sql语句，优化去吧。还等什么呢？

1.54：缺失索引

缺失索引就是帮你查找你的数据库缺少什么索引，告诉你那些字段需要加上索引，这样你就可以根据提示添加你数据库缺少的索引了

SELECT TOP 10

[Total Cost] = ROUND(avg\_total\_user\_cost \* avg\_user\_impact \* (user\_seeks + user\_scans),0)

, avg\_user\_impact

, TableName = statement

, [EqualityUsage] = equality\_columns

, [InequalityUsage] = inequality\_columns

, [Include Cloumns] = included\_columns

FROM    sys.dm\_db\_missing\_index\_groups g

INNER JOIN sys.dm\_db\_missing\_index\_group\_stats s

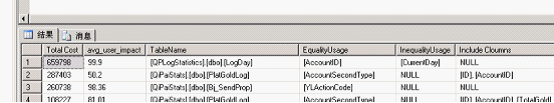
ON s.group\_handle = g.index\_group\_handle

INNER JOIN sys.dm\_db\_missing\_index\_details d

ON d.index\_handle = g.index\_handle

ORDER BY [Total Cost] DESC;

查询结果如下：



**1.6  适当创建索引覆盖**

　　假设你在Sales表(SelesID,SalesDate,SalesPersonID,ProductID,Qty)的外键列(ProductID)上创建了一个索引，假设ProductID列是一个高选中性列，那么任何在where子句中使用索引列(ProductID)的select查询都会更快，如果在外键上没有创建索引，将会发生全部扫描，但还有办法可以进一步提升查询性能。

　　假设Sales表有10,000行记录，下面的SQL语句选中400行(总行数的4%)：

SELECT SalesDate, SalesPersonID FROM Sales WHERE ProductID = 112

　　我们来看看这条SQL语句在SQL执行引擎中是如何执行的：

　　1)Sales表在ProductID列上有一个非聚集索引，因此它查找非聚集索引树找出ProductID=112的记录;

　　2)包含ProductID = 112记录的索引页也包括所有的聚集索引键(所有的主键键值，即SalesID);

　　3)针对每一个主键(这里是400)，SQL Server引擎查找聚集索引树找出真实的行在对应页面中的位置;

　　SQL Server引擎从对应的行查找SalesDate和SalesPersonID列的值。

　　在上面的步骤中，对ProductID = 112的每个主键记录(这里是400)，SQL Server引擎要搜索400次聚集索引树以检索查询中指定的其它列(SalesDate，SalesPersonID)。

　　如果非聚集索引页中包括了聚集索引键和其它两列(SalesDate,，SalesPersonID)的值，SQL Server引擎可能不会执行上面的第3和4步，直接从非聚集索引树查找ProductID列速度还会快一些，直接从索引页读取这三列的数值。

　　幸运的是，有一种方法实现了这个功能，它被称为“覆盖索引”，在表列上创建覆盖索引时，需要指定哪些额外的列值需要和聚集索引键值(主键)一起存储在索引页中。下面是在Sales 表ProductID列上创建覆盖索引的例子：

CREATE INDEX NCLIX\_Sales\_ProductID--Index name  
  
　　ON dbo.Sales(ProductID)--Column on which index is to be created  
　　INCLUDE(SalesDate, SalesPersonID)--Additional column values to include

　　应该在那些select查询中常使用到的列上创建覆盖索引，但覆盖索引中包括过多的列也不行，因为覆盖索引列的值是存储在[内存](http://product.it168.com/list/b/0205_1.shtml)中的，这样会消耗过多内存，引发性能下降。

**1.7 索引碎片**

在数据库性能优化一：数据库自身优化一文中已经讲到了这个问题，再次就不做过多的重复地址：<http://www.cnblogs.com/AK2012/archive/2012/12/25/2012-1228.html>

**1.8 索引实战（摘抄）**

之所以这章摘抄，是因为下面这个文章已经写的太好了，估计我写出来也无法比这个好了，所以就摘抄了

人们在使用SQL时往往会陷入一个误区，即太关注于所得的结果是否正确，而忽略了不同的实现方法之间可能存在的性能差异，这种性能差异在大型的或是复杂的数据库环境中（如联机事务处理OLTP或决策支持系统DSS）中表现得尤为明显。

笔者在工作实践中发现，不良的SQL往往来自于不恰当的索引设计、不充份的连接条件和不可优化的where子句。

在对它们进行适当的优化后，其运行速度有了明显地提高！

下面我将从这三个方面分别进行总结：

为了更直观地说明问题，所有实例中的SQL运行时间均经过测试，不超过１秒的均表示为（< 1秒）。----

测试环境: 主机：HP LH II---- 主频：330MHZ---- 内存：128兆----

操作系统：Operserver5.0.4----

数据库：Sybase11.0.3

一、不合理的索引设计----

例：表record有620000行，试看在不同的索引下，下面几个 SQL的运行情况：

---- 1.在date上建有一非个群集索引

select count(\*) from record where date >'19991201' and date < '19991214'and amount >2000 (25秒)

select date ,sum(amount) from record group by date(55秒)

select count(\*) from record where date >'19990901' and place in ('BJ','SH') (27秒)

---- 分析：----

date上有大量的重复值，在非群集索引下，数据在物理上随机存放在数据页上，在范围查找时，必须执行一次表扫描才能找到这一范围内的全部行。

---- 2.在date上的一个群集索引

select count(\*) from record where date >'19991201' and date < '19991214' and amount >2000 （14秒）

select date,sum(amount) from record group by date（28秒）

select count(\*) from record where date >'19990901' and place in ('BJ','SH')（14秒）

---- 分析：---- 在群集索引下，数据在物理上按顺序在数据页上，重复值也排列在一起，因而在范围查找时，可以先找到这个范围的起末点，且只在这个范围内扫描数据页，避免了大范围扫描，提高了查询速度。

---- 3.在place，date，amount上的组合索引

select count(\*) from record where date >'19991201' and date < '19991214' and amount >2000 （26秒）

select date,sum(amount) from record group by date（27秒）

select count(\*) from record where date >'19990901' and place in ('BJ, 'SH')（< 1秒）

---- 分析：---- 这是一个不很合理的组合索引，因为它的前导列是place，第一和第二条SQL没有引用place，因此也没有利用上索引；第三个SQL使用了place，且引用的所有列都包含在组合索引中，形成了索引覆盖，所以它的速度是非常快的。

---- 4.在date，place，amount上的组合索引

select count(\*) from record where date >'19991201' and date < '19991214' and amount >2000(< 1秒)

select date,sum(amount) from record group by date（11秒）

select count(\*) from record where date >'19990901' and place in ('BJ','SH')（< 1秒）

---- 分析：---- 这是一个合理的组合索引。它将date作为前导列，使每个SQL都可以利用索引，并且在第一和第三个SQL中形成了索引覆盖，因而性能达到了最优。

---- 5.总结：----

缺省情况下建立的索引是非群集索引，但有时它并不是最佳的；合理的索引设计要建立在对各种查询的分析和预测上。

一般来说：

①.有大量重复值、且经常有范围查询（between, >,< ，>=,< =）和order by、group by发生的列，可考虑建立群集索引；

②.经常同时存取多列，且每列都含有重复值可考虑建立组合索引；

③.组合索引要尽量使关键查询形成索引覆盖，其前导列一定是使用最频繁的列。

二、不充份的连接条件：

例：表card有7896行，在card\_no上有一个非聚集索引，表account有191122行，在account\_no上有一个非聚集索引，试看在不同的表连接条件下，两个SQL的执行情况：

select sum(a.amount) from account a,card b where a.card\_no = b.card\_no（20秒）

select sum(a.amount) from account a,card b where a.card\_no = b.card\_no and a.account\_no=b.account\_no（< 1秒）

---- 分析：---- 在第一个连接条件下，最佳查询方案是将account作外层表，card作内层表，利用card上的索引，其I/O次数可由以下公式估算为：

外层表account上的22541页+（外层表account的191122行\*内层表card上对应外层表第一行所要查找的3页）=595907次I/O

在第二个连接条件下，最佳查询方案是将card作外层表，account作内层表，利用account上的索引，其I/O次数可由以下公式估算为：外层表card上的1944页+（外层表card的7896行\*内层表account上对应外层表每一行所要查找的4页）= 33528次I/O

可见，只有充份的连接条件，真正的最佳方案才会被执行。

总结：

1.多表操作在被实际执行前，查询优化器会根据连接条件，列出几组可能的连接方案并从中找出系统开销最小的最佳方案。连接条件要充份考虑带有索引的表、行数多的表；内外表的选择可由公式：外层表中的匹配行数\*内层表中每一次查找的次数确定，乘积最小为最佳方案。

2.查看执行方案的方法-- 用set showplanon，打开showplan选项，就可以看到连接顺序、使用何种索引的信息；想看更详细的信息，需用sa角色执行dbcc(3604,310,302)。

三、不可优化的where子句

1.例：下列SQL条件语句中的列都建有恰当的索引，但执行速度却非常慢：

select \* from record wheresubstring(card\_no,1,4)='5378'(13秒)

select \* from record whereamount/30< 1000（11秒）

select \* from record whereconvert(char(10),date,112)='19991201'（10秒）

分析：

where子句中对列的任何操作结果都是在SQL运行时逐列计算得到的，因此它不得不进行表搜索，而没有使用该列上面的索引；

如果这些结果在查询编译时就能得到，那么就可以被SQL优化器优化，使用索引，避免表搜索，因此将SQL重写成下面这样：

select \* from record where card\_no like'5378%'（< 1秒）

select \* from record where amount< 1000\*30（< 1秒）

select \* from record where date= '1999/12/01'（< 1秒）

你会发现SQL明显快起来！

2.例：表stuff有200000行，id\_no上有非群集索引，请看下面这个SQL：

select count(\*) from stuff where id\_no in('0','1')（23秒）

分析：---- where条件中的'in'在逻辑上相当于'or'，所以语法分析器会将in ('0','1')转化为id\_no ='0' or id\_no='1'来执行。

我们期望它会根据每个or子句分别查找，再将结果相加，这样可以利用id\_no上的索引；

但实际上（根据showplan）,它却采用了"OR策略"，即先取出满足每个or子句的行，存入临时数据库的工作表中，再建立唯一索引以去掉重复行，最后从这个临时表中计算结果。因此，实际过程没有利用id\_no上索引，并且完成时间还要受tempdb数据库性能的影响。

实践证明，表的行数越多，工作表的性能就越差，当stuff有620000行时，执行时间竟达到220秒！还不如将or子句分开：

select count(\*) from stuff where id\_no='0'select count(\*) from stuff where id\_no='1'

得到两个结果，再作一次加法合算。因为每句都使用了索引，执行时间只有3秒，在620000行下，时间也只有4秒。

或者，用更好的方法，写一个简单的存储过程：

create proc count\_stuff asdeclare @a intdeclare @b intdeclare @c intdeclare @d char(10)beginselect @a=count(\*) from stuff where id\_no='0'select @b=count(\*) from stuff where id\_no='1'endselect @c=@a+@bselect @d=convert(char(10),@c)print @d

直接算出结果，执行时间同上面一样快！

---- 总结：---- 可见，所谓优化即where子句利用了索引，不可优化即发生了表扫描或额外开销。

1.任何对列的操作都将导致表扫描，它包括数据库函数、计算表达式等等，查询时要尽可能将操作移至等号右边。

2.in、or子句常会使用工作表，使索引失效；如果不产生大量重复值，可以考虑把子句拆开；拆开的子句中应该包含索引。

3.要善于使用存储过程，它使SQL变得更加灵活和高效。

从以上这些例子可以看出，SQL优化的实质就是在结果正确的前提下，用优化器可以识别的语句，充份利用索引，减少表扫描的I/O次数，尽量避免表搜索的发生。其实SQL的性能优化是一个复杂的过程，上述这些只是在应用层次的一种体现，深入研究还会涉及数据库层的资源配置、网络层的流量控制以及操作系统层的总体设计。

1.7索引实战是摘抄网友的文章，引用地址：<http://blog.csdn.net/gprime/article/details/1687930>

[**利用SQL索引提高查询速度**](http://blog.csdn.net/hank5658/article/details/6075546)

**1.合理使用索引**

索引是[数据库](http://lib.csdn.net/base/mysql)中重要的[数据结构](http://lib.csdn.net/base/datastructure)，它的根本目的就是为了提高查询效率。现在大多数的数据库产品都采用IBM最先提出的*ISAM*索引结构。

索引的使用要恰到好处，其使用原则如下：

在经常进行连接，但是没有指定为外键的列上建立索引，而不经常连接的字段则由优化器自动生成索引。

在频繁进行排序或分组（即进行group by或order by操作）的列上建立索引。

在条件表达式中经常用到的不同值较多的列上建立检索，在不同值少的列上不要建立索引。比如在雇员表的“性别”列上只有“男”与“女”两个不同值，因此就无必要建立索引。如果建立索引不但不会提高查询效率，反而会严重降低更新速度。

如果待排序的列有多个，可以在这些列上建立复合索引（compound index）。

使用系统工具。如Informix数据库有一个tbcheck工具，可以在可疑的索引上进行检查。在一些数据库服务器上，索引可能失效或者因为频繁操作而 使得读取效率降低，如果一个使用索引的查询不明不白地慢下来，可以试着用tbcheck工具检查索引的完整性，必要时进行修复。另外，当数据库表更新大量 数据后，删除并重建索引可以提高查询速度。

(1)在下面两条select语句中:

SELECT \* FROM table1 WHERE field1<=10000 AND field1>=0;   
SELECT \* FROM table1 WHERE field1>=0 AND field1<=10000;

如果数据表中的数据field1都>=0,则第一条select语句要比第二条select语句效率高的多，因为第二条select语句的第一个条件耗费了大量的系统资源。

第一个原则：在where子句中应把最具限制性的条件放在最前面。

(2)在下面的select语句中:

SELECT \* FROM tab WHERE a=… AND b=… AND c=…;

若有索引index(a,b,c)，则where子句中字段的顺序应和索引中字段顺序一致。

第二个原则：where子句中字段的顺序应和索引中字段顺序一致。

——————————————————————————   
以下假设在field1上有唯一索引I1，在field2上有非唯一索引I2。   
——————————————————————————   
(3) SELECT field3,field4 FROM tb WHERE field1='sdf' 快   
SELECT \* FROM tb WHERE field1='sdf' 慢[/cci]

因为后者在索引扫描后要多一步ROWID表访问。

(4) SELECT field3,field4 FROM tb WHERE field1>='sdf' 快   
SELECT field3,field4 FROM tb WHERE field1>'sdf' 慢

因为前者可以迅速定位索引。

(5) SELECT field3,field4 FROM tb WHERE field2 LIKE 'R%' 快   
SELECT field3,field4 FROM tb WHERE field2 LIKE '%R' 慢，

因为后者不使用索引。

(6) 使用函数如：   
SELECT field3,field4 FROM tb WHERE upper(field2)='RMN'不使用索引。

如果一个表有两万条记录，建议不使用函数；如果一个表有五万条以上记录，严格禁止使用函数！两万条记录以下没有限制。

(7) 空值不在索引中存储，所以   
SELECT field3,field4 FROM tb WHERE field2 IS[NOT] NULL不使用索引。

(8) 不等式如   
SELECT field3,field4 FROM tb WHERE field2!='TOM'不使用索引。   
相似地，   
SELECT field3,field4 FROM tb WHERE field2 NOT IN('M','P')不使用索引。

(9) 多列索引，只有当查询中索引首列被用于条件时，索引才能被使用。

(10) MAX，MIN等函数，使用索引。   
SELECT max(field2) FROM tb 所以，如果需要对字段取max，min，sum等，应该加索引。

一次只使用一个聚集函数，如：   
SELECT “min”=min(field1), “max”=max(field1) FROM tb   
不如：SELECT “min”=(SELECT min(field1) FROM tb) , “max”=(SELECT max(field1) FROM tb)

(11) 重复值过多的索引不会被查询优化器使用。而且因为建了索引，修改该字段值时还要修改索引，所以更新该字段的操作比没有索引更慢。

(12) 索引值过大（如在一个char(40)的字段上建索引），会造成大量的I/O开销（甚至会超过表扫描的I/O开销）。因此，尽量使用整数索引。 Sp\_estspace可以计算表和索引的开销。

(13) 对于多列索引，ORDER BY的顺序必须和索引的字段顺序一致。

(14) 在sybase中，如果ORDER BY的字段组成一个簇索引，那么无须做ORDER BY。记录的排列顺序是与簇索引一致的。

(15) 多表联结（具体查询方案需要通过[测试](http://lib.csdn.net/base/softwaretest)得到）   
where子句中限定条件尽量使用相关联的字段，且尽量把相关联的字段放在前面。   
SELECT a.field1,b.field2 FROM a,b WHERE a.field3=b.field3

field3上没有索引的情况下:   
对a作全表扫描，结果排序   
对b作全表扫描，结果排序   
结果合并。   
对于很小的表或巨大的表比较合适。

field3上有索引   
按照表联结的次序，b为驱动表，a为被驱动表   
对b作全表扫描   
对a作索引范围扫描   
如果匹配，通过a的rowid访问

(16) 避免一对多的join。如：   
SELECT tb1.field3,tb1.field4,tb2.field2 FROM tb1,tb2 WHERE tb1.field2=tb2.field2 AND tb1.field2=‘BU1032’ AND tb2.field2= ‘aaa’   
不如：   
declare @a varchar(80)   
SELECT @a=field2 FROM tb2 WHERE field2=‘aaa’   
SELECT tb1.field3,tb1.field4,@a FROM tb1 WHERE field2= ‘aaa’

(16) 子查询   
用exists/not exists代替in/not in操作   
比较：   
SELECT a.field1 FROM a WHERE a.field2 IN(SELECT b.field1 FROM b WHERE b.field2=100)   
SELECT a.field1 FROM a WHERE EXISTS( SELECT 1 FROM b WHERE a.field2=b.field1 AND b.field2=100)   
SELECT field1 FROM a WHERE field1 NOT IN( SELECT field2 FROM b)   
SELECT field1 FROM a WHERE NOT EXISTS( SELECT 1 FROM b WHERE b.field2=a.field1)

(17) 主、外键主要用于数据约束，sybase中创建主键时会自动创建索引，外键与索引无关，提高性能必须再建索引。

(18) char类型的字段不建索引比int类型的字段不建索引更糟糕。建索引后性能只稍差一点。

(19) 使用count(\*)而不要使用count(column\_name)，避免使用count(DISTINCT column\_name)。

(20) 等号右边尽量不要使用字段名，如：   
SELECT \* FROM tb WHERE field1 = field3

(21) 避免使用or条件，因为or不使用索引。

**2.避免使用order by和group by字句。**

因为使用这两个子句会占用大量的临时空间(tempspace),如果一定要使用，可用视图、人工生成临时表的方法来代替。   
如果必须使用，先检查memory、tempdb的大小。   
测试证明，特别要避免一个查询里既使用join又使用group by，速度会非常慢！

**3.尽量少用子查询，特别是相关子查询。因为这样会导致效率下降。**

一个列的标签同时在主查询和where子句中的查询中出现，那么很可能当主查询中的列值改变之后，子查询必须重新查询一次。查询嵌套层次越多，效率越低，因此应当尽量避免子查询。如果子查询不可避免，那么要在子查询中过滤掉尽可能多的行。

**4．消除对大型表行数据的顺序存取**

在 嵌套查询中，对表的顺序存取对查询效率可能产生致命的影响。   
比如采用顺序存取策略，一个嵌套3层的查询，如果每层都查询1000行，那么这个查询就要查询 10亿行数据。   
避免这种情况的主要方法就是对连接的列进行索引。   
例如，两个表：学生表（学号、姓名、年龄……）和选课表（学号、课程号、成绩）。如果两个 表要做连接，就要在“学号”这个连接字段上建立索引。   
还可以使用并集来避免顺序存取。尽管在所有的检查列上都有索引，但某些形式的where子句强迫优化器使用顺序存取。   
下面的查询将强迫对orders表执行顺序操作：   
SELECT ＊ FROM orders WHERE (customer\_num=104 AND order\_num>1001) OR order\_num=1008   
虽然在customer\_num和order\_num上建有索引，但是在上面的语句中优化器还是使用顺序存取路径扫描整个表。因为这个语句要检索的是分离的行的集合，所以应该改为如下语句：   
SELECT ＊ FROM orders WHERE customer\_num=104 AND order\_num>1001   
UNION   
SELECT ＊ FROM orders WHERE order\_num=1008   
这样就能利用索引路径处理查询。

**5．避免困难的正规表达式**

MATCHES和LIKE关键字支持通配符匹配，技术上叫正规表达式。但这种匹配特别耗费时间。例如：SELECT ＊ FROM customer WHERE zipcode LIKE “98\_ \_ \_”   
即使在zipcode字段上建立了索引，在这种情况下也还是采用顺序扫描的方式。如果把语句改为SELECT ＊ FROM customer WHERE zipcode >“98000”，在执行查询时就会利用索引来查询，显然会大大提高速度。   
另外，还要避免非开始的子串。例如语句：SELECT ＊ FROM customer WHERE zipcode[2，3] >“80”，在where子句中采用了非开始子串，因而这个语句也不会使用索引。

**6．使用临时表加速查询**

把表的一个子集进行排序并创建临时表，有时能加速查询。它有助于避免多重排序操作，而且在其他方面还能简化优化器的工作。例如：   
SELECT cust.name，rcvbles.balance，……other COLUMNS   
FROM cust，rcvbles   
WHERE cust.customer\_id = rcvlbes.customer\_id   
AND rcvblls.balance>0   
AND cust.postcode>“98000”   
ORDER BY cust.name   
如果这个查询要被执行多次而不止一次，可以把所有未付款的客户找出来放在一个临时文件中，并按客户的名字进行排序：   
SELECT cust.name，rcvbles.balance，……other COLUMNS   
FROM cust，rcvbles   
WHERE cust.customer\_id = rcvlbes.customer\_id   
AND rcvblls.balance>;0   
ORDER BY cust.name   
INTO TEMP cust\_with\_balance   
然后以下面的方式在临时表中查询：   
SELECT ＊ FROM cust\_with\_balance   
WHERE postcode>“98000”   
临时表中的行要比主表中的行少，而且物理顺序就是所要求的顺序，减少了磁盘I/O，所以查询工作量可以得到大幅减少。   
注意：临时表创建后不会反映主表的修改。在主表中数据频繁修改的情况下，注意不要丢失数据。

**7．用排序来取代非顺序存取**

非顺序磁盘存取是最慢的操作，表现在磁盘存取臂的来回移动。SQL语句隐藏了这一情况，使得我们在写应用程序时很容易写出要求存取大量非顺序页的查询。