数据库相关

# 一、游标

## 定义

游标其实可以理解成一个定义在特定数据集上的指针，我们可以控制这个指针遍历数据集，或者仅仅是指向特定的行，所以游标是定义在以Select开始的数据集上的。

在关系数据库中，我们对于查询的思考是面向集合的。而游标打破了这一规则，游标使得我们思考方式变为逐行进行.对于类C的开发人员来着，这样的思考方式会更加舒服。

T-SQL中的游标定义在MSDN中如下:

|  |
| --- |
| DECLARE cursor\_name CURSOR [ LOCAL | GLOBAL ]  [ FORWARD\_ONLY | SCROLL ]  [ STATIC | KEYSET | DYNAMIC | FAST\_FORWARD ]  [ READ\_ONLY | SCROLL\_LOCKS | OPTIMISTIC ]  [ TYPE\_WARNING ]  FOR select\_statement  [ FOR UPDATE [ OF column\_name [ ,...n ] ] ]  [;] |

**LOCAL和GLOBAL二选一**

LOCAL意味着游标的生存周期只在批处理或函数或存储过程中可见，而GLOBAL意味着游标对于特定连接作为上下文，全局内有效。如果不指定游标作用域，默认作用域为GLOBAL。

**FORWARD\_ONLY 和 SCROLL 二选一**

FORWARD\_ONLY意味着游标只能从数据集开始向数据集结束的方向读取，FETCH NEXT是唯一的选项，而SCROLL支持游标在定义的数据集中向任何方向，或任何位置移动

**STATIC KEYSET DYNAMIC 和 FAST\_FORWARD 四选一**

这四个关键字是游标所在数据集所反映的表内数据和游标读取出的数据的关系。

STATIC意味着，当游标被建立时，将会创建FOR后面的SELECT语句所包含数据集的**副本**存入tempdb数据库中，任何对于**底层**表内数据的更改**不会影响**到游标的内容.

DYNAMIC是和STATIC完全相反的选项,当底层数据库更改时，游标的内容也**随之得到反映**，在下一次fetch中，数据内容会随之改变

KEYSET可以理解为介于STATIC和DYNAMIC的**折中**方案。将游标所在结果集的唯一能确定每一行的主键存入tempdb,当结果集中任何行改变或者删除时，@@FETCH\_STATUS会为-2,KEYSET无法探测新加入的数据

FAST\_FORWARD可以理解成FORWARD\_ONLY的优化版本.FORWARD\_ONLY执行的是静态计划，而FAST\_FORWARD是根据情况进行选择采用动态计划还是静态计划，大多数情况下FAST\_FORWARD要比FORWARD\_ONLY性能略好.

**READ\_ONLY SCROLL\_LOCKS OPTIMISTIC 三选一**

READ\_ONLY意味着声明的游标只能读取数据,游标不能做任何更新操作。

SCROLL\_LOCKS是另一种极端，将读入游标的所有数据进行锁定，防止其他程序进行更改，以确保更新的绝对成功

OPTIMISTIC是相对比较好的一个选择，OPTIMISTIC不锁定任何数据，当需要在游标中更新数据时,如果底层表数据更新，则游标内数据更新不成功，如果底层表数据未更新，则游标内表数据可以更新。

## 语法

### 打开

当定义完游标后，游标需要打开后使用，只有简单一行代码:

|  |
| --- |
| OPEN test\_Cursor |

注意，当全局游标和局部游标变量重名时，默认会打开局部变量游标

### 使用

游标的使用分为两部分,一部分是操作游标在数据集内的指向，另一部分是将游标所指向的行的部分或全部内容进行操作。

只有支持6种移动选项,分别为到第一行（FIRST),最后一行(LAST),下一行(NEXT),上一行(PRIOR),直接跳到某行(ABSOLUTE(n)),相对于目前跳几行(RELATIVE(n))

对于未指定SCROLL选项的游标来说，只支持NEXT取值。

第一步操作完成后，就通过INTO关键字将这行的值传入局部变量。

游标经常会和全局变量@@FETCH\_STATUS与WHILE循环来共同使用,以达到遍历游标所在数据集的目的

### 关闭

在游标使用完之后，一定要记得关闭,只需要一行代码:CLOSE+游标名称

|  |
| --- |
| CLOSE test\_Cursor |

### 释放

当游标不再需要被使用后，释放游标，只需要一行代码:DEALLOCATE+游标名称

|  |
| --- |
| DEALLOCATE test\_Cursor |

## 建议

对于游标一些优化建议

* 如果能不用游标，尽量不要使用游标
* 用完用完之后一定要关闭和释放
* 尽量不要在大量数据上定义游标
* 尽量不要使用游标上更新数据
* 尽量不要使用insensitive, static和keyset这些参数定义游标
* 如果可以，尽量使用FAST\_FORWARD关键字定义游标
* 如果只对数据进行读取，当读取时只用到FETCH NEXT选项，则最好使用FORWARD\_ONLY参数

使用游标经常会比使用面向集合的方法慢2-3倍，当游标定义在大数据量时，这个比例还会增加。如果可能，尽量使用while,子查询，临时表，函数，表变量等来替代游标，记住，游标永远只是你最后无奈之下的选择，而不是首选。

参考： <http://www.cnblogs.com/moss_tan_jun/archive/2011/11/26/2263988.html>

25).尽量避免使用游标，

因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

26).使用基于游标的方法或临时表方法之前，

应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27).与临时表一样，游标并不是不可使用。

对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28).在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。

无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

# 二、索引

## 定义

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**。

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。

**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

## 建立原则

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；

10) 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；

11) 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.最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

2.=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

3.尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录

4.索引列不能参与计算，保持列“干净”，比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create\_time = unix\_timestamp(’2014-05-29’);

5.尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可

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。

## 使用

### 创建

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。

### 删除

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。

### 显示索引信息

使用系统存储过程：sp\_helpindex tablename查看指定表的索引信息。

## 索引覆盖

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中的，这样会消耗过多内存，引发性能下降**。

<http://www.cnblogs.com/AK2012/archive/2013/01/04/2844283.html>

索引存储结构：<http://www.cnblogs.com/ASPNET2008/p/3325532.html>

# 视图

视图提供了存储**预定义的查询语句**作为数据库中的对象以备以后使用的能力；视图只是一种**逻辑对象**，并不是物理对象，因为视图不占物理存储空间；在视图中被查询的表称为视图的基表；视图只在刚刚打开的一瞬间,通过定义从基表中搜集数据,并展现给用户。

优点：

1. 集中用户使用的数据；2．掩码数据库的复杂性，视图把数据库设计的复杂性与用户屏蔽分开；3．简化用户权限的管理；可以实现对表字段子集的访问权限4．为向其他应用程序输出而重新组织数据。

创建视图：

CREATE VIEW dbo.vw\_cjd(name, cid, result)

AS

SELECT name, report.cid, report.result FROM student JOIN report ON student.sid=report.sid

显示视图：

Select \* from vw\_cjd

修改视图：

ALTER VIEW vw\_cjd

AS

Select name, report.cid, report.result, address From student join report ON student.sid=report.sid

删除视图：

DROP VIEW vw\_cjd

通过视图修改数据只能影响一个基表；如果指定WITH CHECK OPTION选项，那么要验证所修改的数据。

更新视图与更新表格一样,但是在视图中使用了多个基本表连接的情况下,每次更新操作只能更新来自基本表的一个数据列

我们知道视图是依赖于表,如果在一个表中创建一个视图,今后如果这个表被删除了,则这个视图将不可再用了.为了防止用户删除一个有视图在引用的表,可以在创建视图的时候加上schemabinding关键字.

视图与查询的区别：

　　视图和查询都是用由sql语句组成,这是他们相同的地方,但是视图和查询有着本质区别：

　　它们的区别在于：

　　1：存储上的区别：视图存储为数据库设计的一部分,而查询则不是.

　　2：更新限制的要求不一样

　　要注意：因为视图来自于表,所以通过视图可以**间接对表进行更新**,我们也可以通过update语句对表进行更新,但是对视图和查询更新限制是不同的,以下我们会知道虽然通过视图可以间接更新表但是有很多限制.

3：排序结果：通过sql语句,可以对一个表进行排序,而视图则不行。

### 索引视图

the official name of this is 'Indexed View', Not 'View Index'.

在视图上创建的第一个索引必须是唯一聚集索引。在创建唯一聚集索引后，可创建其它非聚集索引。视图上的索引命名规则与表上的索引命名规则相同。唯一区别是表名由视图名替换。

执行 CREATE INDEX 语句的用户必须是视图的所有者。

当执行 CREATE INDEX 语句时，下列 SET 选项必须设置为 ON：

ANSI\_NULLS

ANSI\_PADDING

ANSI\_WARNINGS

ARITHABORT

CONCAT\_NULL\_YIELDS\_NULL

QUOTED\_IDENTIFIERS

必须将选项 NUMERIC\_ROUNDABORT 选项设置为 OFF。

视图不能包含 text、ntext 或 image 列，即使在 CREATE INDEX 语句中没有引用它们。

如果视图定义中的 SELECT 语句指定了一个 GROUP BY 子句，则唯一聚集索引的键只能引用在 GROUP BY 子句中指定的列。

### 物化视图

缺点：非实时

<http://www.cnblogs.com/GISDEV/archive/2008/02/13/1067817.html>

存储过程

所谓可编程数据库，其“可编程”往往体现在存储过程的创建上。存储过程是存储在数据库中的一组**已被编译**的命令集合，创建时已经预编译和**内部优化**，一般具有**较高的执行速度**（相比于独立执行这些命令）。存储过程的另一种表现是**函数**，即带有返回值的过程。

触发器是一种特殊类型的存储过程，当用户修改、删除或插入数据时，这种特殊存储过程就会自动触发执行。

## SQL语句优化

何为执行计划

执行计划是数据库根据SQL语句和相关表的统计信息作出的一个查询方案，这个方案是由查询优化器自动分析产生的，比如一条SQL语句如果用来从一个 10万条记录的表中查1条记录，那查询优化器会选择“索引查找”方式，如果该表进行了归档，当前只剩下5000条记录了，那查询优化器就会改变方案，采用 “全表扫描”方式。

可见，执行计划并不是固定的，它是“个性化的”。产生一个正确的“执行计划”有两点很重要：

(1) SQL语句是否清晰地告诉查询优化器它想干什么？

(2) 查询优化器得到的数据库统计信息是否是最新的、正确的？

**统一SQL语句的写法**

对于以下两句SQL语句，程序员认为是相同的，数据库查询优化器认为是不同的。

|  |
| --- |
| select\*from dual  select\*From dual |

其实就是大小写不同，查询分析器就认为是两句不同的SQL语句，必须进行两次解析。生成2个执行计划。所以作为程序员，应该保证相同的查询语句在任何地方都一致，多一个空格都不行！

**不要把SQL写得太复杂**

一般，将一个Select语句的结果作为子集，然后从该子集中再进行查询，这种一层嵌套语句还是比较常见的，但是根据经验，超过3层嵌套，查询优化器就很容易给出错误的执行计划。因为它被绕晕了。像这种类似人工智能的东西，终究比人的分辨力要差些，如果人都看晕了，我可以保证数据库也会晕的。

另外，执行计划是可以被重用的，越简单的SQL语句被重用的可能性越高。而复杂的SQL语句只要有一个字符发生变化就必须重新解析，然后再把这一大堆垃圾塞在内存里。可想而知，数据库的效率会何等低下。

**临时表暂存中间结果**

简化SQL语句的重要方法就是采用临时表暂存中间结果，但是，临时表的好处远远不止这些，将临时结果暂存在临时表，后面的查询就在tempdb中了，这可以避免程序中多次扫描主表，也大大减少了程序执行中“共享锁”阻塞“更新锁”，减少了阻塞，提高了并发性能。

除非你有一个很特别的原因去使用 NULL 值，你应该总是让你的字段保持 NOT NULL。不要以为 NULL 不需要空间，其需要额外的空间，并且，在你进行比较的时候，你的程序会更复杂。

**只在必要的情况下才使用begin tran**

SQL Server中**一句SQL语句默认就是一个事务**，在该语句执行完成后也是默认commit的。其实，这就是begin tran的一个**最小化**的形式，好比在每句语句开头隐含了一个begin tran，结束时隐含了一个commit。

有些情况下，我们需要显式声明begin tran，比如做“插、删、改”操作需要同时修改几个表，要求要么几个表都修改成功，要么都不成功。begin tran 可以起到这样的作用，它可以把若干SQL语句套在一起执行，最后再一起commit。好处是保证了数据的一致性，但任何事情都不是完美无缺的。Begin tran付出的**代价**是在提交之前，所有SQL语句**锁住的资源**都不能释放，直到commit掉。

可见，如果Begin tran套住的SQL语句太多，那数据库的性能就糟糕了。在该大事务提交之前，必然会阻塞别的语句，造成block很多。

Begin tran使用的原则是，在保证数据一致性的前提下，begin tran 套住的SQL语句**越少越好**！有些情况下可以采用**触发器同步数据**，不一定要用begin tran。

**一些SQL查询语句应加上nolock**

在SQL语句中加nolock是提高SQL Server并发性能的重要手段，在oracle中并不需要这样做，因为oracle的结构更为合理，有undo表空间保存“数据前影”，该数据如果在修改中还未commit，那么你读到的是它修改之前的副本，该副本放在undo表空间中。这样，oracle的读、写可以做到互不影响，这也是oracle 广受称赞的地方。SQL Server 的读、写是会相互阻塞的，为了提高并发性能，对于一些查询，可以加上nolock，这样读的时候可以允许写，但缺点是可能读到未提交的脏数据。使用 nolock有3条原则。

(1) 查询的结果用于“插、删、改”的不能加nolock ！

(2) 查询的表属于频繁发生页分裂的，慎用nolock ！

(3) 使用临时表一样可以保存“数据前影”，起到类似oracle的undo表空间的功能，

能采用临时表提高并发性能的，不要用nolock 。

**尽量早过滤**

这一优化策略其实最常见于索引的优化设计中（将过滤性更好的字段放得更靠前）。

在 SQL 编写中同样可以使用这一原则来优化一些 Join 的 SQL。比如我们在多个表进行分页数据查询的时候，我们最好是能够在一个表上先过滤好数据分好页，然后再用分好页的结果集与另外的表 Join，这样可以尽可能多的减少不必要的 IO 操作，大大节省 IO 操作所消耗的时间。

### 关于WHERE

a.避免在WHERE子句中使用in，not in，or 或者having。可以使用 **exist 和not exist代替 in和not in**。

可以使用表链接代替 exist。Having可以用where代替，如果无法代替可以分两步处理。

|  |
| --- |
| SELECT \* FROM ORDERS WHERE CUSTOMER\_NAME NOT IN  (SELECT CUSTOMER\_NAME FROM CUSTOMER)  优化为：  SELECT \* FROM ORDERS WHERE CUSTOMER\_NAME not exist  (SELECT CUSTOMER\_NAME FROM CUSTOMER) |

可以用合并查询**union代替or**

b.不要以字符格式声明数字，要以数字格式声明字符值。（日期同样）否则会使索引无效，产生全表扫描。

|  |
| --- |
| SELECT emp.ename, emp.job FROM emp WHERE emp.empno = 7369;  不要使用：SELECT emp.ename, emp.job FROM emp WHERE emp.empno = ‘7369’ |

c.OLTP系统必须使用绑定变量

|  |
| --- |
| select\*from orderheader where changetime >'2010-10-20 00:00:01'  select\*from orderheader where changetime >'2010-09-22 00:00:01' |

以上两句语句，查询优化器认为是不同的SQL语句，需要解析两次。如果采用绑定变量

|  |
| --- |
| select\*from orderheader where changetime >@chgtime |

@chgtime变量可以传入任何值，这样大量的类似查询可以重用该执行计划了，这可以大大降低数据库解析SQL语句的负担。一次解析，多次重用，是提高数据库效率的原则。

d.绑定变量窥测

事物都存在两面性，绑定变量对大多数OLTP处理是适用的，但是也有例外。比如在where条件中的字段是“**倾斜字段**”的时候。

“倾斜字段”指该列中的绝大多数的值都是相同的，比如一张人口调查表，其中“民族”这列，90%以上都是汉族。那么如果一个SQL语句要查询30岁的汉族人口有多少，那“民族”这列必然要被放在where条件中。这个时候如果采用绑定变量@nation会存在很大问题。

试想如果@nation传入的第一个值是“汉族”，那整个执行计划必然会选择表扫描。然后，第二个值传入的是“布依族”，按理说“布依族”占的比例可能只有万分之一，应该采用索引查找。但是，由于重用了第一次解析的“汉族”的那个执行计划，那么第二次也将采用表扫描方式。这个问题就是著名的“绑定变量窥测”，建议对于“倾斜字段”不要采用绑定变量。

e.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

### 关于索引

**使用like进行模糊查询时应注意，**比如

|  |
| --- |
| select\*from contact where username like ‘%yue%’ |

关键词%yue%，由于yue**前面**用到了“%”，因此该查询**必然走全表扫描**，除非必要，否则**不要在关键词前加%**

前缀索引才能使用索引

使用**否定方式检索**（如NOT 和！=），将强制进行**全表扫描**，忽略所有的索引。

对于大型表，**范围扫描**通常比全表扫描更加有效，因为某些类型的索引，如B树索引，可以进行有效的范围查找。

尽量使用数字或较短的字符串作为**索引字段**；单字段索引相比复合字段索引占用空间更少，与表空间相差更大，对I/O的减少作用更明显。满足参照完整性的表如果缺少外键索引将会严重影响性能，因此有时考虑为性能牺牲参照完整性。

避免对索引字段进行**计算**操作

避免在索引列上使用IS **NULL**和IS NOT NULL

避免在索引列上出现**数据类型转换**

避免在索引字段上使用**函数**

避免建立索引的列中使用**空值**。

如果你的应用程序有很多 **JOIN 查询**，你应该确认两个表中Join的字段是被建过索引的。这样，MySQL内部会启动为你优化Join的SQL语句的机制。

而且，这些被用来Join的字段，应该是相同的类型的。例如：如果你要把 DECIMAL 字段和一个 INT 字段Join在一起，MySQL就无法使用它们的索引。对于那些STRING类型，还需要有相同的字符集才行。（两个表的字符集有可能不一样）

**聚集索引没有建在表的顺序字段上，该表容易发生页分裂**

比如订单表，有订单编号orderid，也有客户编号contactid，那么聚集索引应该加在哪个字段上呢？对于该表，订单编号是顺序添加的，如果在orderid上加聚集索引，新增的行都是添加在末尾，这样不容易经常产生页分裂。然而，由于大多数查询都是根据客户编号来查的，因此，将聚集索引加在contactid上才有意义。而contactid对于订单表而言，并非顺序字段。

比如“张三”的“contactid”是001，那么“张三”的订单信息必须都放在这张表的第一个数据页上，如果今天“张三”新下了一个订单，那该订单信息不能放在表的最后一页，而是第一页！如果第一页放满了呢？很抱歉，该表所有数据都要往后移动为这条记录腾地方。

**加nolock后查询经常发生页分裂的表，容易产生跳读或重复读**

加nolock后可以在“插、删、改”的同时进行查询，但是由于同时发生“插、删、改”，在某些情况下，一旦该数据页满了，那么页分裂不可避免，而此时nolock的查询正在发生，比如在第100页已经读过的记录，可能会因为页分裂而分到第101页，这有可能使得nolock查询在读101页时重复读到该条数据，产生“重复读”。同理，如果在100页上的数据还没被读到就分到99页去了，那nolock查询有可能会漏过该记录，产生“跳读”。

**数据类型的隐式转换对查询效率的影响**

sql server2000的数据库，我们的程序在提交sql语句的时候，没有使用强类型提交这个字段的值，由sql server 2000自动转换数据类型，会导致传入的参数与主键字段类型不一致，这个时候sql server 2000可能就会使用全表扫描。Sql2005上没有发现这种问题，但是还是应该注意一下。

**SQL Server 表连接的三种方式**

(1) Merge Join

(2) Nested Loop Join

(3) Hash Join

SQL Server 2000只有一种join方式——Nested Loop Join，如果A结果集较小，那就默认作为外表，A中每条记录都要去B中扫描一遍，实际扫过的行数相当于A结果集行数x B结果集行数。所以如果两个结果集都很大，那Join的结果很糟糕。

SQL Server 2005新增了Merge Join，如果A表和B表的连接字段正好是聚集索引所在字段，那么表的顺序已经排好，只要两边拼上去就行了，这种join的开销相当于A表的结果集行数加上B表的结果集行数，一个是加，一个是乘，可见merge join 的效果要比Nested Loop Join好多了。

如果连接的字段上没有索引，那SQL2000的效率是相当低的，而SQL2005提供了Hash join，相当于临时给A，B表的结果集加上索引，因此SQL2005的效率比SQL2000有很大提高，我认为，这是一个重要的原因。

总结一下，在表连接时要注意以下几点：

(1) 连接字段尽量选择聚集索引所在的字段

(2) 仔细考虑where条件，尽量减小A、B表的结果集

(3) 如果很多join的连接字段都缺少索引，而你还在用SQL Server 2000，赶紧升级吧。

检索特定的字段名比用**’\*’**检索所有字段更为有效，因为‘\*’检索需要先在元数据字典中查找元数据，以找到表中的字段，从而增加了额外的开销，在高并发的数据库中，连续的数据字典查询可能严重影响数据库的响应性能。

**Like**运算符效率不高，通常倾向于全表扫描。

**很小的表**直接查询可能比用索引更快

**在可以使用UNION ALL的语句里，使用了UNION**

UNION 因为会将各查询子集的记录做比较，故比起UNION ALL ，通常速度都会慢上许多。一般来说，如果使用UNION ALL能满足要求的话，务必使用UNION ALL。还有一种情况大家可能会忽略掉，就是虽然要求几个子集的并集需要过滤掉重复记录，但由于脚本的特殊性，不可能存在重复记录，这时便应该使用UNION ALL

避免使用耗费资源的操作，带有DISTINCT,UNION,MINUS,INTERSECT,ORDER BY的SQL语句会启动SQL引擎 执行，耗费资源的排序(SORT)功能. DISTINCT需要一次排序操作, 而其他的至少需要执行两次排序

**尽量用 union all 代替 union**

union 和 union all 的差异主要是前者需要将两个（或者多个）结果集合并后再进行唯一性过滤操作，这就会涉及到排序，增加大量的 CPU 运算，加大资源消耗及延迟。所以当我们可以**确认不可能出现重复结果集或者不在乎重复结果集**的时候，尽量使用 union all 而不是 union。

count(\*) 优于count(1)和count(primary\_key)，count(column) 和 count(\*) 是不一样的

这个误区甚至在很多的资深工程师或者是 DBA 中都普遍存在，很多人都会认为这是理所当然的。实际上，count(column) 和 count(\*) 是一个完全不一样的操作，所代表的意义也完全不一样。count(column) 是表示结果集中有多少个column字段不为空的记录，　　count(\*) 是表示整个结果集有多少条记录

**尽量用 join 代替子查询**

虽然 Join 性能并不佳，但是和 MySQL 的子查询比起来还是有非常大的性能优势。MySQL 的子查询执行计划一直存在较大的问题，虽然这个问题已经存在多年，但是到目前已经发布的所有稳定版本中都普遍存在，一直没有太大改善。

**知道结果的记录数量时**

当你查询表的有些时候，你已经知道结果只会有一条结果，但因为你可能需要去fetch游标，或是你也许会去检查返回的记录数。

在这种情况下，加上 LIMIT 1 可以增加性能。这样一样，MySQL数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据。

**用>=替代>**

高效:

SELECT \* FROM EMP WHERE DEPTNO >=4

低效:

SELECT \* FROM EMP WHERE DEPTNO >3

两者的区别在于, 前者DBMS将直接跳到第一个DEPT等于4的记录而后者将首先定位到DEPTNO=3的记录并且向前扫描到第一个DEPT大于3的记录.

来源：<http://www.cnblogs.com/atree/archive/2011/02/13/sql_optimize_1.html>

<http://database.51cto.com/art/200904/118526.htm>

<http://blog.csdn.net/hguisu/article/details/5731629> 高级篇

<http://coolshell.cn/articles/1846.html>

<http://isky000.com/database/mysql-performance-tuning-sql>

慢查询优化 <http://tech.meituan.com/mysql-index.html>

<http://blog.itpub.net/235507/viewspace-707862/>

## 事务

## 存储过程

## 临时表

尽量使用表变量来代替临时表。

如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

避免频繁创建和删除临时表，以减少系统表资源的消耗。

临时表并不是不可使用，

适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；

如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

常见的**避免临时表**的方法有：

1）创建索引：在ORDER BY或者GROUP BY的列上创建索引；

2）分拆很长的列：一般情况下，TEXT、BLOB，大于512字节的字符串，基本上都是为了显示信息，而不会用于查询条件， 因此表设计的时候，应该将这些列独立到另外一张表。

常见的优化SQL语句方法如下：

1）拆分SQL语句

临时表主要是用于**排序和分组**，很多业务都是要求排序后再取出详细的分页数据，这种情况下可以将排序和取出详细数据拆分成不同的SQL，以降低排序或分组时临时表的大小，提升排序和分组的效率，我们的案例就是采用这种方法。

2）优化业务，去掉排序分组等操作

有时候业务其实并不需要排序或分组，仅仅是为了好看或者阅读方便而进行了排序，例如数据导出、数据查询等操作，这种情况下去掉排序和分组对业务也没有多大影响。