作者：沙漠　QQ：41796745

1 数据表结构的设计与性能优化

1.1 、数据表的存储原理

SQL Server 每次读取 1 个存储块，每个存储块大小为 8KB，每读取 1 个存储块计算为 1 个逻辑读。
问题：如果数据内容非常大，像我们系统中的 Feeling 字段非常大，就会导致每个存储块存放的数据行数会非常少，这样当我们读取数据时，要读取许多的存储块。

存储块 1（8KB）

聚集索引	数据内容
聚集索引	数据内容
聚集索引	数据内容
聚集索引	数据内容

存储块 2（8KB）

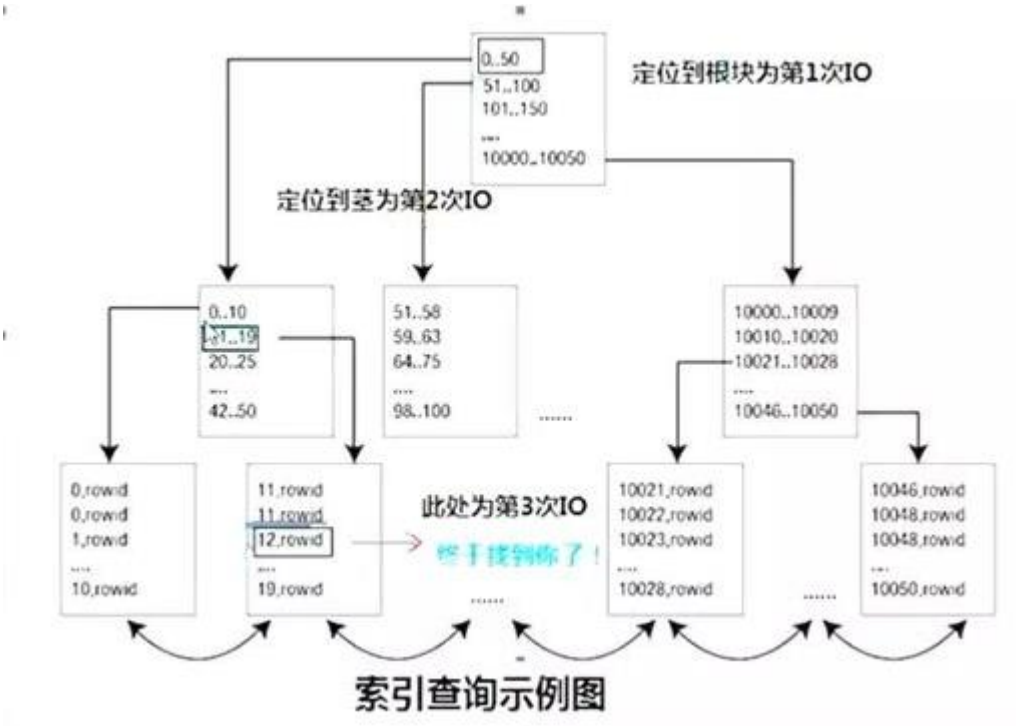
聚集索引	数据内容
聚集索引	数据内容
聚集索引	数据内容
聚集索引	数据内容

1.2 表设计的优化

1.2.1、 字段类型优先级

Bit>int，int>date,time>char,varchar>text，原因：整型，time 运算快，节省空间。
所以我们在表设计时，如果是 bool 类型的数据值就不应该用 int 类型字段。

1.2.2、 聚集索引字段类型的选择



主键的索引结构中，即存储了主键值，又存储了行数据，这种结构称为“聚集索引”
在插入数据时，数据节点会分裂，这个问题比较严重，节点下存储了“行数据”，分裂的时候，还要移动行数据。如果主键是无规律的，则会加速它的数据节点分裂，且效率极低。

高性能索引策略：
主键，尽量用整型，而且是递增的整型。如果是无规律的数据，将会产生页的分裂，影响速度。索引长度直接影响索引文件大小，影响增删改的速度，并间接影响查询速度（占用内存多）。因为主键索引在物理存放时是有序的，如果主键的值是无序的，那么主键每次插入时，索引文件都重新进行排序，会产生额外的数据消耗，另外主键的叶子上存放的数据，还会导致叶子数据行的移动和分裂，又会产生一些消耗，所以主键尽量用整型，且自增的类型。

1.2.3、 纵向拆分

纵向拆分方法：把不需要用于查询的大字段，放到另外一个新建的附属表中，如 feelingStructured 表和 Auto 表。这样就将 Evaluation 表的数据内容减少到最少，存储块中可以多存储许多数据行，减少程序读取存储块的个数。

1.2.4、 横向拆分

横向拆分方法：表分区，表分区的条件，一张数据表的行数至少要达 3000W 行以上的数据，就可以考虑做表分区了。但这不是绝对，如果表的数据行内容特别多，查询特别慢时，也可以尽早做表分区。
注意问题：普通表在查询时，会比分区表要快一些，因为基于分区表的查询会遍历所有的分区表，而普通表只查询了普通表一个表。
解决办法，在查询条件中加入分区条件，这样查询就会落入指定的分区中，不用遍历所有的分区，但问题是，是不是所有的查询都能加入分区条件呢。只要进行了表分区，那么 SQL 的前提条件就是所有 SQL 都要加上分区条件，除非个别的汇总，统计类的 SQL。

1.2.5、 数据库分库

数据库分库：当一个台数据库表服务器访问压力过大，数据量过大时，就需要考虑进行数据库分库，数据库分库条件和表分区的逻辑是比较像的。根据业务条件，如地区，时间，进行拆分。

1.2.6、 读索引为什么比读表快

这里面引发出一个触类旁通的问题，为什么索引查询会比直接查数据要快？因为索引做为一个独立的数据存储区，也是跟数据表存储块一样，以 8KB 为一个存储块，一个 IO 读取一次存储块，而索引中只有简单的几个索引列，而不是整个数据行的数据，所以它一个 IO 读取的数据会非常多，这样它的 IO 就会非常少，加快了查询速度。

1.2.7、 数据压缩的利与弊

数据压缩和索引压缩会使存储空间和逻辑读减少，但是会使表更新的开销加大，查询耗费的 CPU 也更多，所以压缩表一般适合在更新比较少，且 CPU 消耗不大，IO 消耗很大系统中适用。像企业管理软件就比较适用于数据压缩和 BI 系统。如果当前系统的 IO 并不高，但 CPU 非常繁忙，则不应该采用表和索引压缩，传统数据库的压缩率并不是太高，真正压缩率比较高的应该是 BI 的数据。

2 索引优化

2.1 索引列的设计与优化

索引列数据的重复度称为可选择性，如性别列的取值范围为”男，女”，这个索引列可选择性就比较低，你在里面能找出太多相同的数据列出来，如选择列数据内容为唯一的，则可选

择性非常高。我们选择索引列时，要尽量选择高选择性的列。
案例分析：现在有一个商家黑盒处罚功能，假如商家发布违规的商品和促销信息，则会被系统管理员设置为 5 天或 10 天的墨盒状态，墨盒状态期间无法发布商品和促销信息，现在设计有商家黑盒表，有 3 字段，BusinessID（商家 ID）， 生效开始时间（StartTime）、生效结束时间（EndTime），这生效开始时间和生效结束时间 2 个列谁的选择性高呢？
示例业务数据：

BusinessID	StartTime	EndTime
11111	2014-3-1	2014-3-5
223333	2015-4-8	2015-4-13-
23423424	2016-1-13	2016-1-18

现在找出今天还属于黑盒的商家名单列表

Where startTime<=’2016-1-14’ and EndTime>=’2016-1-14’

大家可以看到，符合 startTime<=’2016-1-14’的有 3 条记录，而符合 EndTime>=’2016-1-14’的只有 1 条记录，所以 EndTime 的可选择性比 StartTime 高，这就会决定，我们到底是采用哪一列做索引列，如果 2 个列都做索引列，哪个索引列会排在前面（多列索引的设计见下面）。

2.2 表扫描查询如何修改为索引列扫描

如：我们项目中以前用来判断用户是否填写 Feeling 时，用 where len(feeling)>0，len(best)>0，Len(wrost)>0，
在 where 条件中用函数会导致表扫描，所以应该设计成标识字段，如 IsEditFeeling(bit)，IsWroteBest(bit)，IsWroteWrost(bit)，SQL 就可以优化成 where IsEditFeeling=1，IsWrostBest=1，IsWroteWrost=1。这样就是索引扫描。

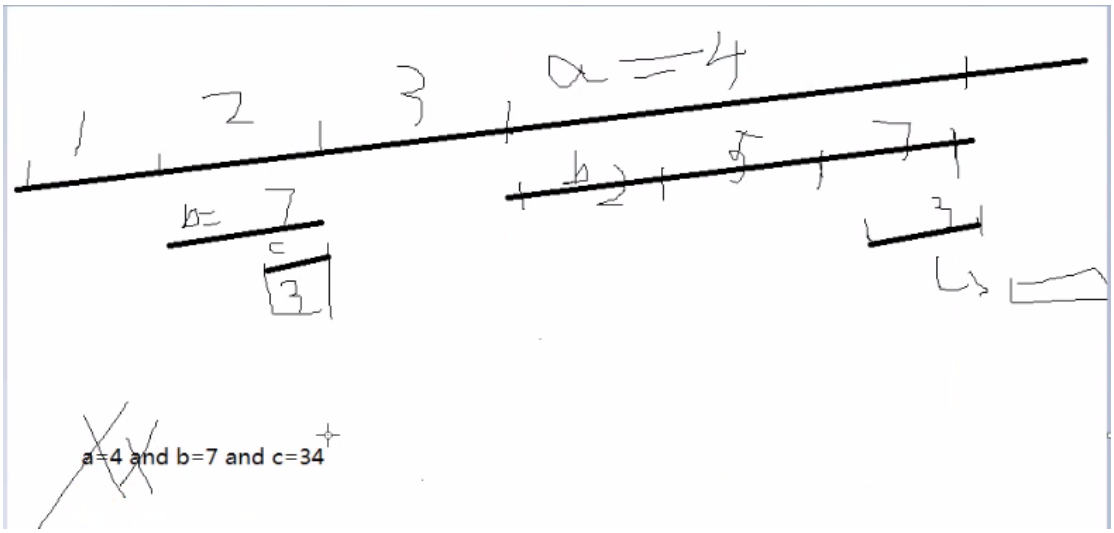
2.3 单列索引的设计与优化

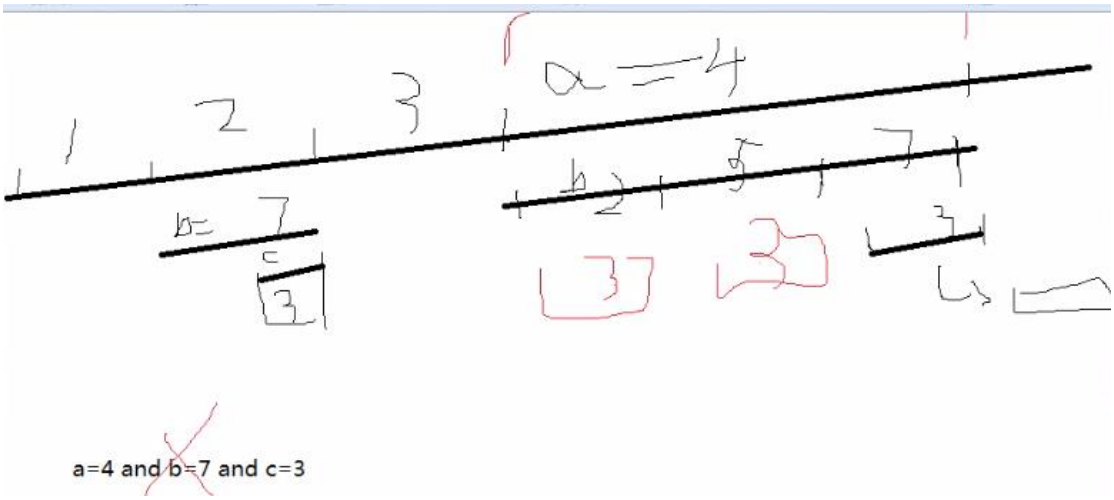
单列索引设计比较简单，一般就是根据业务条件来定义就行，如根据车系 ID 或车型 ID。但要注意索引列的可选择性。

2.4 多列索引的设计与优化

2.4.1、 多列索引的存储规则

假设某个表有一个联合索引(a,b,c)，我们来看下在这个索引中是如何存储这些字段数据的。





2.4.2、 多列索引左前缀规则

多列索引必须用到第 1 个，否则不生效。

2.2 在多列上建立索引后,查询哪个列,索引都将发挥作用
误: 多列索引上,索引发挥作用,需要满足左前缀要求.
以 index(a,b,c) 为例,

语句	索引是否发挥作用
Where a=3	是,只使用了 a 列
Where a=3 and b=5	是,使用了 a,b 列
Where a=3 and b=5 and c=4	是,使用了 abc
Where b=3 或 where c=4	否
Where a=3 and c=4	a 列能发挥索引,c 不能
Where a=3 and b>10 and c=7	A 能利用,b 能利用, C 不能利用
同上,where a=3 and b like 'xxxx%' and c=7	A 能用,B 能用,C 不能用

2.4.3、 多列索引列的选择优化

根据以上的多列索引列的存储规则和左前缀规则，在建多列索引时，应该将可选择项高的列放在最左边，后面依次类推，如上面的黑盒案例分析中，应该将 endTime 列放在最左边，然后才是 starttime 列。

在选择性相同的情况下，应该把等值（如：=）的放在左边，不等值（如：>，<）放在后面。

```
Select * from t where object_id>=20 andobject_id<2000 and object_type=’ TABLE’ ;
```

这个查询对应的索引，应该建成

```
Create index idx_id_type on t(object_type,object_id);
```

2.5 书签查找优化

2.5.1、 数据表物理存储和索引的物理索引

非聚簇索引的另一个好处是，它有一个独立于数据表的结构，所以可以被放置在不同的文件组，使用不同的 I/O 路径，这在第 2 章中介绍过。这意味着 SQL Server 可以并行访问索引和表，使查找更快速。

数据存储块 1（8KB）

聚集索引	数据内容
聚集索引	数据内容
聚集索引	数据内容
聚集索引	数据内容

数据存储块 2（8KB）

聚集索引	数据内容
聚集索引	数据内容
聚集索引	数据内容
聚集索引	数据内容

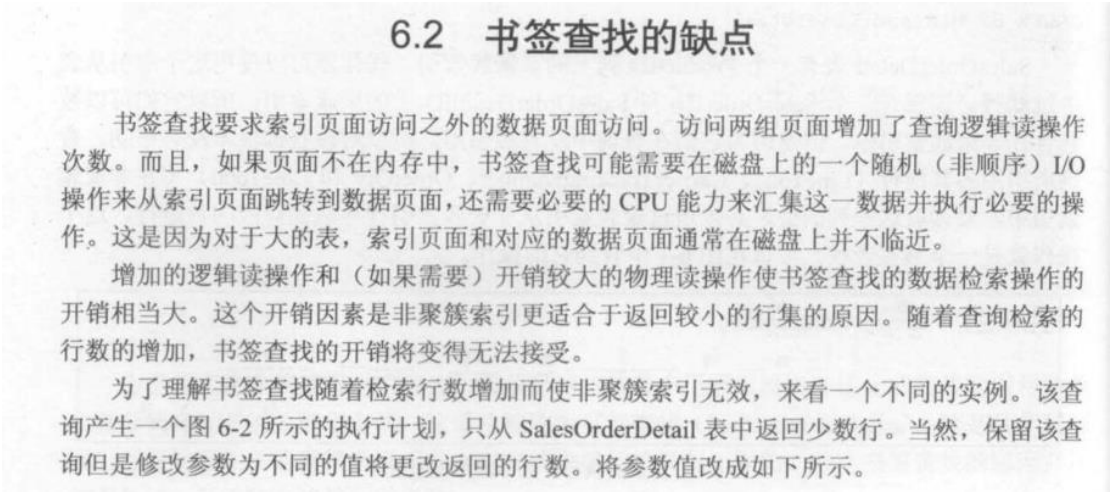
索引存储块 1（8KB）

非聚集索引 1(include(区))	非聚集索引 2	非聚集索引 N

非聚集索引 1	非聚集索引 2	非聚集索引 N
非聚集索引 1	非聚集索引 2	非聚集索引 N
非聚集索引 1	非聚集索引 2	非聚集索引 N

因为索引里面只存储了固定了几列，如果查询时需要读取索引列之外的数据，就需要到数据存储块，根据聚集索引去重新查找我们想要的数据，这就叫书签查找。

2.5.2、 书签查找的缺点

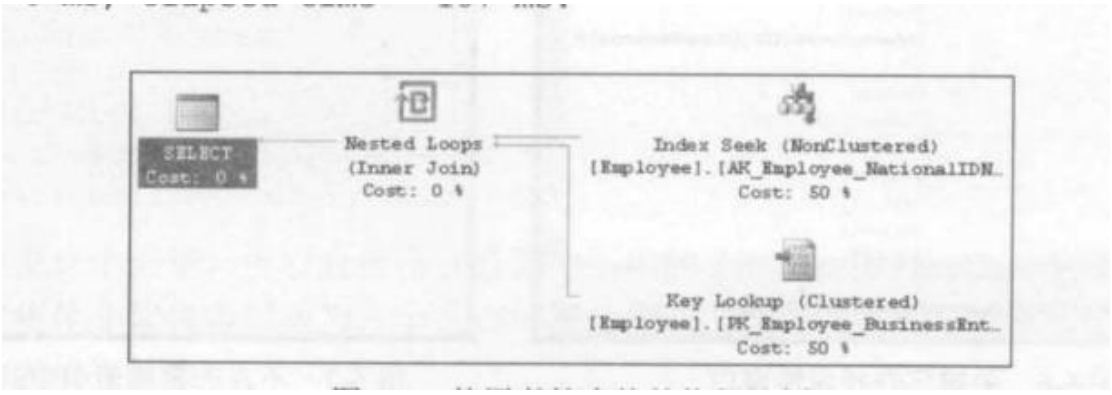


6.2 书签查找的缺点

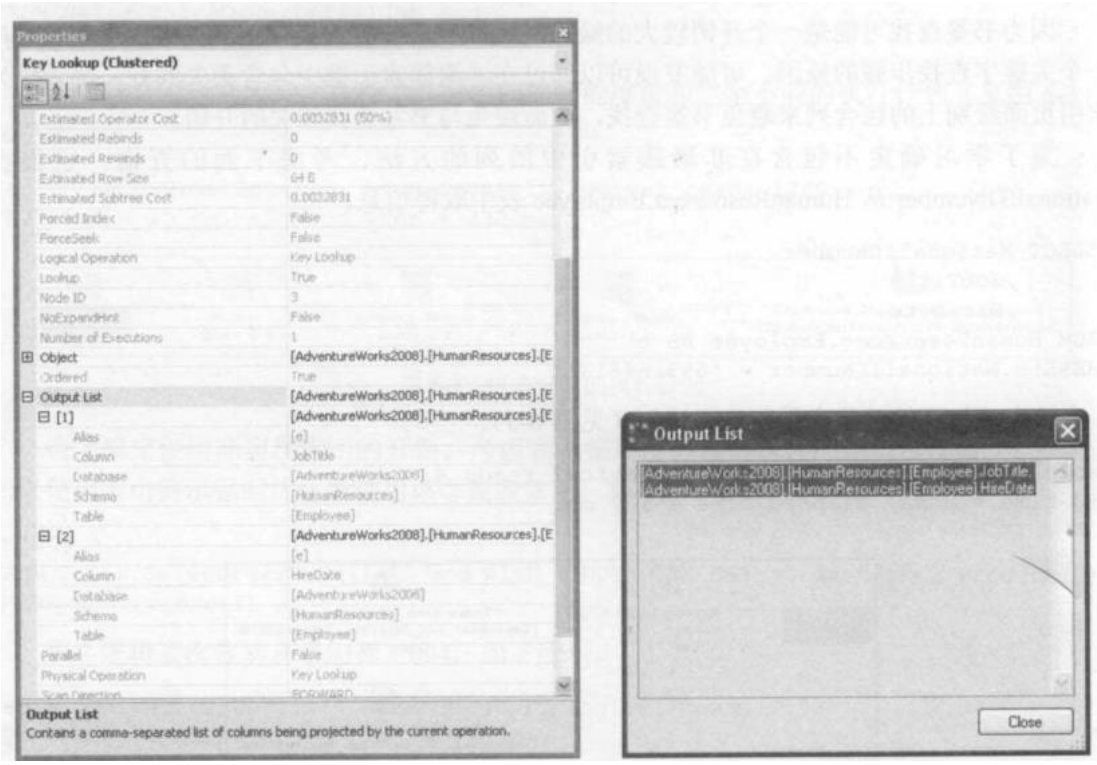
书签查找要求索引页面访问之外的数据页面访问。访问两组页面增加了查询逻辑读操作次数。而且，如果页面不在内存中，书签查找可能需要在磁盘上的一个随机（非顺序）I/O操作来从索引页面跳转到数据页面，还需要必要的 CPU 能力来汇集这一数据并执行必要的操作。这是因为对于大的表，索引页面和对应的数据页面通常在磁盘上并不临近。

增加的逻辑读操作和（如果需要）开销较大的物理读操作使书签查找的数据检索操作的开销相当大。这个开销因素是非聚簇索引更适合于返回较小的行集的原因。随着查询检索的行数的增加，书签查找的开销将变得无法接受。

为了理解书签查找随着检索行数增加而使非聚簇索引无效，来看一个不同的实例。该查询产生一个图 6-2 所示的执行计划，只从 SalesOrderDetail 表中返回少数行。当然，保留该查询但是修改参数为不同的值将更改返回的行数。将参数值改成如下所示。



找出书签查找中查找的列



2.5.3、 书签查找优化

6.4.1 使用一个聚簇索引

对于聚簇索引，索引的叶子页面和表的数据页面相同。因此，当读取聚簇索引键列的值时，数据引擎可以读取其他列的值而不需要任何导航。在前一个实例中，如果将非聚簇索引转换为一个特定行的聚簇索引，SQL Server 可以从相同的页面读取所有列。

把非聚簇索引转换为一个聚簇索引说起来很简单。但是，在这个例子和大部分可能遇到的情况下，这不可能做到，因为表已经有了一个聚簇索引。这个表的聚簇索引恰好是主键。必须卸载所有外键约束，卸载并且重建主键为一个非聚簇索引，然后重新创建 NationalIDNumber 上的索引。不仅需要考虑所涉及的工作，还可能严重地影响依赖于现有聚簇索引的其他查询。

注意 记住，一个表只能有一个聚簇索引。

6.4.2 使用一个覆盖索引

第 4 章中介绍了一个覆盖索引类似于一个用于查询的伪聚簇索引，因为它可以返回结果而不需要求助于表数据。所以，也可以使用覆盖索引来避免书签查找。

为了理解使用覆盖索引避免书签查找的方法，再次研究对 HumanResources.Employee 表的查询，如下所示。

```
SELECT NationalIDNumber
      ,JobTitle
      ,HireDate
FROM   HumanResources.Employee AS e
WHERE  e.NationalIDNumber = '693168613'
```

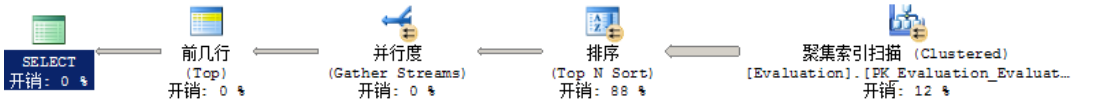
为了避免这个书签，可以直接将该查询中引用的 JobTitle 和 HireDate 列添加到非聚簇索引键中。这将使非聚簇索引成为这个查询的覆盖索引，因为所有列可以从索引中检索而不需要转到基本表或聚簇索引。

2.6 索引与排序的优化

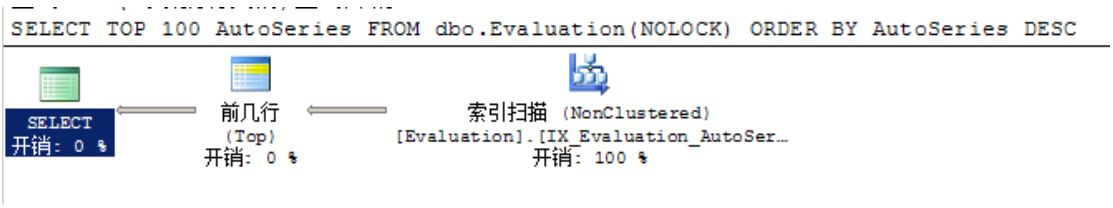
索引在物理存储上是有序的，所以如果我们的 SQL 的排序是基于索引列进行的，那么不需要再重新进行排序，反之，系统会在 temp 库中建立表变量或临时表，然后在这个表变量或临时表中重新进行排序。如果想为多表联查时，基于多个表做排序字段建立索引，如 Order by A.Id,B.Created，则可以基于 2 个表建立一个物化视图，然后在这个物化视图上为这 2 列建立索引。

```
Create index idx_t on t(col1 desc,col2 asc)

Select * from t order by col1 desc,col2 asc

0：sorts(memory)

0：sorts(disk)
```

```
SELECT TOP 100 * FROM dbo.Evaluation(NOLOCK) ORDER BY Created DESC
```




```
SQL Server 执行时间:
   CPU 时间= 2182 毫秒，占用时间= 231 毫秒。
SQL Server 分析和编译时间:

   CPU 时间= 0 毫秒，占用时间= 0 毫秒。
```




```
SQL Server 执行时间:
   CPU 时间= 0 毫秒，占用时间= 2 毫秒。
SQL Server 分析和编译时间:

   CPU 时间= 0 毫秒，占用时间= 0 毫秒。
```

2.7 关于索引的坏处

在有几个索引和情况下和只有主键索引的情况下，插入数据的速度相差 10 倍，而且数据表里数据记录越大，插入速度越明显。
在无索引的情况下，表的记录越大，插入的速度只会受到很小的影响，基本不会越慢。
在无索引的情况下，分区表的插入要比普通表更慢，因为插入的数据需要做判断，有这方面的开销。
解决办法：读写分离，在主库（或叫写库）上只有主键索引，而没有别的索引。只读库上有许多用于查询的索引，然后写库的数据定时同步到只读库上，这样的话，插入时不会因为索引的原因导致插入变慢，也不会因为没有索引导致查询变慢。

2.8 索引建立时的开销及注意事项

建索引过程会产生全表锁和建索引过程中会产生全表排序.
后果：普通的对表建索引将会导致针对该表的更新操作无法进行，需要等待索引建完，更新操作将会被建索引的动作阻塞。

解决办法：

```
CREATE NONCLUSTERED INDEX [IX_Auto_Serial] ON [dbo].[Auto]
(
    [Serial] ASC,
    [RowStatus] ASC
)
 WITH (  ONLINE = ON)
```

在创建索引时，加上 online 参数，这种建索引的方式不会阻止针对该表的更新操作，与建普通索引相反的是，online 建索引的动作是反过来被更新操作阻塞。

3 多表联查优化

3.1 、字段冗余，不要联查

业务案例分析：在口碑表里冗余新建一个 LastAppendingDrivenKilometers 字段，这样就不用关联追加表进行查询了。再冗余新建一个 AutoBoughtCity 字段，就不用关联 Auto 表进行查询了。

3.2 、多表联查的实现原理及表关联字段的设计原则

三大表连接的概要说明：1、Nested Loops Join。2、Hash Join。3、Merge Sort Join
Nested Loops Join 驱动结果集的条数决定被驱动表的访问次数
Hash Join 两表各自只会访问 1 次或 0 次。
Merge Sort Join 与 Hash Join 的相同

表驱动顺序与性能（Nested LoopsJoin 性能与驱动顺序有关）
Hash Join 性能与驱动顺序有关（和 NL 相似）
Merge Sort Join 性能与表驱动顺序无关

可以用伪代码表示嵌套循环连接算法。

```
for each row R1 in the outer table
   for each row R2 in the inner table
      if R1 joins with R2
         return (R1, R2)
```

```
SELECT *
FROM T1 INNER JOIN T2 ON P1(T1,T2)
        INNER JOIN T3 ON P2(T2,T3)
WHERE P(T1,T2,T3)
```

```
FOR each row t1 in T1 {
 FOR each row t2 in T2 such that P1(t1,t2) {
  FOR each row t3 in T3 such that P2(t2,t3) {
   IF P(t1,t2,t3) {
     t:=t1||t2||t3; OUTPUT t;
   }
  }
 }
}
```

Nested Loops Join 优化要点

1：驱动表的限制条件要有索引。2、被驱动表限制条件要建立索引。3：确保小结果集先驱动，大的被驱动。

Hash Join 忧化要点:1:请确保用在全扫描的 OLAP 场景。2:明确该 SQL 是否限制 Hash Join。3、两表无任何索引倾向 HashJoin。第 1 斧：两表限制条件有索引（看返回量）。第 2 斧：要小结果集先驱动，大的被驱动。第 3 斧：尽量保证 PGA 能容纳 Hash 运算。

Merge Sort Join 优化第 1 式（两表限制条件有索引）
Merge Sort Join 优化第 2 式（连接条件索引消除排序，即消除 2 边分别排序，再合并的这种情况）

因为被驱动表的查询是依赖与 ON 条件中写的字段列，如： A join B on A.CID=B.C ID，
那么应该 B 表的 CID 设置为索引列，最好设置为聚集索引列，这样关联 SQL 在运行时，从 B 表中查找数据时，可以命中索引，否则在 B 表中查找数据时，会引发表扫描和 Hash 匹配。

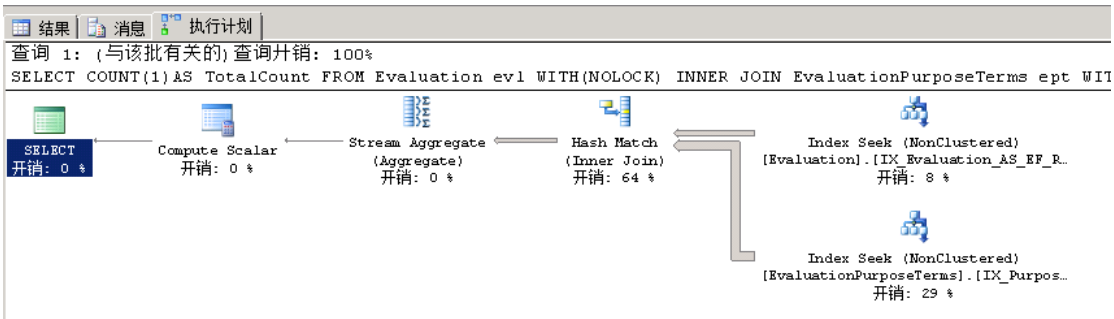
代码示例：

优化前的效果

(1 row(s) affected)
表 'Worktable'。扫描计数 0，逻辑读取 0 次，物理读取 0 次，预读 0 次，lob 逻辑读取 0 次，lob 物理读取 0 次，lob 预读 0 次。
表 'EvaluationPurposeTerms'。扫描计数 1，逻辑读取 1383 次，物理读取 3 次，预读 2269 次，lob 逻辑读取 0 次，lob 物理读取 0 次，lob 预读 0 次。
表 'Evaluation'。扫描计数 1，逻辑读取 110 次，物理读取 2 次，预读 80 次，lob 逻辑读取 0 次，lob 物理读取 0 次，lob 预读 0 次。

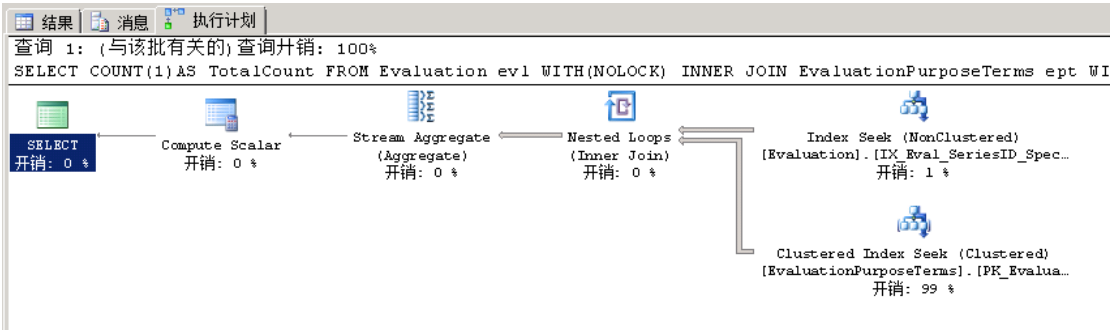
(1 row(s) affected)

SQL Server 执行时间:
CPU 时间 = 140 毫秒，占用时间 = 155 毫秒。



优化后的 SQL:
(1 row(s) affected)
表 'EvaluationPurposeTerms'。扫描计数 0，逻辑读取 1638 次，物理读取 0 次，预读 0 次，

lob 逻辑读取 0 次，lob 物理读取 0 次，lob 预读 0 次。
表 'Evaluation'。扫描计数 1，逻辑读取 20 次，物理读取 0 次，预读 0 次，lob 逻辑读取 0 次，lob 物理读取 0 次，lob 预读 0 次。
SQL Server 执行时间:
CPU 时间 = 0 毫秒，占用时间 = 4 毫秒。



3.3 、关联查询的优化（使用表扫描）

QualityProblem 是一个数据表，QualityProblemDictionary 是一个字典表，字典表的数据基本上是固定不变的，大概 80 行的样子。而在与 QualityProblem 表关联时，QualityProblem 表中符合条件数据的，每一行都会读取一次 QualityProblemDictionary 字典表。SQL 和运行效果如下，现在打算将 QualityProblemDictionary 字典表一次全读取到内存中，这样就不用每条符合条件的 QualityProblem 表中数据都去访问 QualityProblemDictionary 字典表了，修改方法就是在表的 Hint 中指出不要用索引扫描，要采用表扫描。代码：with(index(0))

```
SELECT [QualityId] ,[ItemId] AS [ID] ,d.Name ,d.FullName ,d.FullPath ,d.Level ,d.ParentId FROM [QualityProblem] as p with(nolock)
join [QualityProblemDictionary] as d with(nolock) on d.Id  = p.ItemId where p.QualityId = 592805
```

优化前效果：

```
(7 row(s) affected)
表'QualityProblemDictionary'。扫描计数7，逻辑读取14 次，物理读取0 次，预读0 次，lob 逻辑读取0 次，lob 物理读取0 次，lob 预读0 次。
表'QualityProblem'。扫描计数1，逻辑读取3 次，物理读取0 次，预读0 次，lob 逻辑读取0 次，lob 物理读取0 次，lob 预读0 次。

 SQL Server 执行时间:
   CPU 时间= 15 毫秒，占用时间= 23 毫秒。
```

```
as d with(index(0),nolock) on d.Id  = p.ItemId
```

优化后效果：

```
(7 row(s) affected)
表'Worktable'。扫描计数0，逻辑读取0 次，物理读取0 次，预读0 次，lob 逻辑读取0 次，lob 物理读取0 次，
```

```
lob 预读0 次。
表'QualityProblemDictionary'。扫描计数1，逻辑读取19 次，物理读取0 次，预读0 次，lob 逻辑读取0 次，
lob 物理读取0 次，lob 预读0 次。
表'QualityProblem'。扫描计数1，逻辑读取3 次，物理读取0 次，预读0 次，lob 逻辑读取0 次，lob 物理读取
0 次，lob 预读0 次。

 SQL Server 执行时间:
   CPU 时间= 0 毫秒，占用时间= 2 毫秒。
```

4 查询计划的缓存与 CPU 高优化

4.1 、概述

我们 SQL 执行分为 2 部分，1 是 SQL 查询计划编译部分，2 是 SQL 运行部分。SQL 分析器里显示的 CPU 时间也是包括编译时间和运行时间。

```
SQL Server 分析和编译时间:
    CPU 时间= 0 毫秒，占用时间= 0 毫秒。
SQL Server 执行时间:
   CPU 时间= 2668 毫秒，占用时间= 786 毫秒。
```

因为生成 SQL 的查询计划是非常耗时和耗 CPU 的，SQL Server 在生成查询计划之后，会将已经生成的查询计划保存到计划缓存中，下次再执行该 SQL 时不用再重新生成一个全新的计划。

4.2 、如何使查询计划被缓存

4.2.1、 sp_executesql 存储过程

普通的即席 SQL 的查询计划是没有被缓存的，通过下面的示例可以看出来。

```
USE Northwind2
GO
ALTER DATABASE Northwind2 SET PARAMETERIZATION FORCED
GO
SET STATISTICS IO ON
GO
DBCC FREEPROCCACHE
GO
SELECT * FROM BigOrders WHERE CustomerID =  'CENTC'
GO
SELECT * FROM BigOrders WHERE CustomerID =  'SAVEA'
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
  CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype =  'Compiled Plan'
  AND [text] NOT LIKE  '%dm_exec_cached_plans%' ;
GO
ALTER DATABASE Northwind2 SET PARAMETERIZATION SIMPLE
GO
```

Usecounts	Cacheobjtype	Objtype	Text
1	Compiled Plan	Adhoc	SELECT * FROM BigOrders WHERE CustomerID = 'SAVEA'
1	Compiled Plan	Adhoc	SELECT * FROM BigOrders WHERE CustomerID = 'CENTC'

通过把 SQL 改写为用 sp_executesql 存储过程，即可实现查询计划被缓存。ado.net 底层就是把我们写的 SQL 全部生成 sp_executesql sql 这种格式的 sql，但 ado.net 无法自动区分我们 sql 中哪些是参数，哪些不是参数，所以光靠 ado.net 帮我们自动实现 sp_executesql 还无法实现查询计划的缓存，还需要我们手动把查询条件参数化写完整。最终实现一个标准的语法 Sp_executesql @sqltext,@param1,,,,,,@paramN

这是此过程的通常语法：

```
sp_executesql @batch_text, @batch_parameter_definitions,
  param1,...paramN
```

使用*@batch_text* 和*@batch_parameter_definitions* 同样值的重复调用，通过新指定的参数值使用相同的被缓存的计划。只要计划没有从缓存中删除并且计划是有效的，它就会被

```
USE Northwind2;
GO
SET STATISTICS IO ON;
GO
DBCC FREEPROCCACHE;
GO
EXEC sp_executesql N' SELECT * FROM BigOrders
   WHERE CustomerID = @p' , N' @p nvarchar(10)' ,  'CENTC' ;
GO
EXEC sp_executesql N' SELECT * FROM BigOrders
   WHERE CustomerID = @p' , N' @p nvarchar(10)' ,  'SAVEA' ;
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
   CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype =  'Compiled Plan'
   AND [text] NOT LIKE  '%dm_exec_cached_plans%' ;
GO
SET STATISTICS IO OFF;
GO
```

Usecounts	Cacheobjtype	Objtype	Text
2	Compiled Plan	Prepared	(@p nvarchar(10))SELECT * FROM BigOrders WHERE CustomerID = @p

4.2.2、 存储过程和函数

存储过程和函数的查询计划也会被缓存。

4.3 、如何解决查询计划不被缓存

只要采用 sp_executesql，存储过程，函数这 3 种方式写的 sql 查询计划就会被缓存，但也有原因会导致他们不会被缓存，原因如下：

结果显示，由于一个实际数值的改变，相同的计划不能被重用。但为了利用 adhoc 查询计划的重用，需要确定查询不只是使用了相同的数值，而且查询每个字符都是等同的。如果某个查询有另一个查询没有的新行或额外空白，它们就不会被当做相同的查询。如果某个查询包含其他查询没有的注释，它们也不是等同的。此外，即使在一个大小写不敏感的数据库中，查询也不会是相同的。运行下面的代码，我们会看到所有的查询都不能重用相同的计划。

```
USE Northwind2;
DBCC FREEPROCCACHE;
GO
SELECT * FROM orders WHERE customerID =  'HANAR'
GO
-- Try it again
SELECT * FROM orders WHERE customerID =  'HANAR'
GO
SELECT * FROM orders
WHERE customerID =  'HANAR' ;
GO
SELECT * FROM Orders WHERE CustomerID =  'HANAR'
GO
select * from orders where customerid =  'HANAR'
```

4.3.1、 查询计划不被缓存案例示警

上次线上数据库 CPU 总是高，从 DMV 中查到编译数过高，原因是开发人员为线上跟踪代码运行结果和查错方便，在公共数据访问层中，自动为每一条 SQL 中都加了一条注释，该注释内容为当前系统的 Url，因为一个公共方法会被不同模块调用，当前模块的 Url 自然也是不停的在变，导致 SQL 的查询计划无法重用。

4.4 、缓存的查询计划什么时候被重新生成

1、当表数据增长率达到当前数据量的 30%时，查询计划会被重编译。
2、当新建和修改索引时。
3、当新建和删除字段时。

4.5 、In 条件参数化

Declare @User1 int;Declare @User2 int;Declare @User3 int;Declare @User4 int;declare @User5 int;
Where userId in(@User1,@User2,@User3,@User4,@User5)

5 并发和阻塞问题

1、减少事务长度。
2、SQL 语句加 Nolock。

3、读写分离

6 日志库优化

当有 insert 和 update,delete 操作时，会在日志库中记录日志，随着时间的积累，日志库记录越来越多，每插入一条日志都非常耗时，直接影响我们系统的数据操作。

解决办法：设定每天晚上自动备份，在数据库备份之后，数据库会自动收缩日志库，删除一些日志数据，这样日志库的数据量就下来了。因为数据库觉得你已经备份过了，万一出什么问题，可以用备份文件来恢复，也不需要依靠日志库来进行还原了，没必要再保存那么多日志了。

7 SQLServer 服务器系统参数配置与性能优化

7.1 、CPU 最大并行度

详细描述文档：http://jimshu.blog.51cto.com/3171847/1266978

当系统有多个可用的处理器时，默认情况下 SQL Server 将在并行执行中使用所有处理器。为了更好地控制机器上的负载，可能发现限制并行执行使用的处理器数量是有用的。进一步，可能需要设置相关性，这样某些处理器被保留给操作系统和其他在 SQL Server 之外运行的服务。OLTP 系统经常从完全禁用并行性中获益。

7.2 、并行性开销阀值

在具有多个处理器的系统上，可以并行执行查询。并行性的默认值为 5，这表示优化器估算的开销为查询执行 5 秒。在大部分情况下，这个值太低，意味着越高的并行性值产生越好的性能。在你的系统上测试将有助于确定合适的值。

7.3 、优化即席工作负载

7.4 、数据库文件布局

将数据库文件分别存放在不同的服务器上，以加快并行处理速度。

8 DB 服务器监控

参考附件

数据性能调校--查出最耗资源的各种 SQL.docx

9 SQL 语句性能优化

9.1 、SQL 语句分析方法

SQL 分析技巧，SQL 可以分为 3 段：

1、 Select 部分，重点关注 Select 部分有没有标题子查询，有没有自定义函数
2、 From 后面，重点关注有没有内联视图，有没有视图，有没有进行视图合并
3、 Where 条件部分，看有没有 In/Not In，Exists/Not in 子查询，有没有外连接，有没有在列上面有函数导致不能走索引，比如：where len(feeling)>0
内联视图要手工运行返回多少行，子查询也要查看返回多少行。

9.2 、子查询优化

子查询的特点：主查询返回一行，子查询就会被执行一次。In 语句外面是驱动表，in 语句里面是被驱动表。

优化目的：

1、 子查询不用执行很多次
2、 优化器可以根据统计信息来选择不同的连接方法和不同的连接顺序
子查询中的连接条件，过滤条件分别变成了父查询的连接条件，过滤条件，优化器可以对这些条件进行下推，以提高执行效率。

9.2.1、 非关联子查询

非关联子查询的执行，不依赖于外层父查询的任何属性值。这样子查询具有独立性，可独自求解，形成一个子查询计划先于外层的查询求解，如：

```
SELECT * FROM t1 WHERE col_1 = ANY
  (SELECT col_1 FROM t2 WHERE t2.col_2 = 10);
//子查询语句中（t2）不存在父查询（t1）的属性
```

相关和非相关子询的判断条件就是看子查询是否出现外层父查询中的数据列。

9.2.2、 关联子查询

相关子查询，子查询的执行依赖于外层父查询的一些属性值。子查询因依赖于父查询的参数，当父查询的参数改变时，子查询需要根据新参数值重新执行（查询优化器对相关子查询进行优化有一定意义），如：

```
SELECT * FROM t1 WHERE col_1 = ANY
  (SELECT col_1 FROM t2 WHERE t2.col_2 = t1.col_2);
/* 子查询语句中存在父查询的t1表的col_2列 */
```

9.2.3、 子查询优化方法一（子查询合并）

子查询合并（subquery coalescing）
在某些条件下（语义等价：两个查询块产生同样的结果集），多个子查询能够合并成一个子查询（合并后还是子查询，以后可以通过其他技术消除掉子查询）。这样可以把多次表扫描，多次连接减少为单次表扫描和单次连接，如：

```
SELECT * FROM t1 WHERE a1<10 AND (
   EXISTS (SELECT a2 FROM t2 WHERE t2.a2<5 AND t2.b2=1) OR
   EXISTS (SELECT a2 FROM t2 WHERE t2.a2<5 AND t2.b2=2)
);
```

可优化为：

```
SELECT * FROM t1 WHERE a1<10 AND (
  EXISTS (SELECT a2 FROM t2 WHERE t2.a2<5 AND (t2.b2=1 OR t2.b2=2)
  /*两个ESISTS子句合并为一个，条件也进行了合并*/
);
```

9.2.4、 非关联子查询优化--子查询展开

子查询展开（subquery unnesting）。

又称子查询反嵌套，又称为子查询上拉。
把一些子查询置于外层的父查询中，作为连接关系与外层父查询并列，其实质是把某些子查询重写为等价的多表连接操作（展开后，子查询不存在了，外部查询变成了多表连接）。
带来的好处是，有关的访问路径，连接方法和连接顺序可能被有效使用，使得查询语句的层次尽可能的减少。
WHERE evl.[AutoSeries] =@id and evl.EvaluationID in (select EvaluationID from EvaluationPurposeTerms ept with(nolock) where ept.PurposeID=1)

修改为
INNER JOIN EvaluationPurposeTerms ept ON evl.EvaluationID = ept.EvaluationID
WHERE evl.[AutoSeries] =@id and ept.PurposeID=1

注意：其实子查询和联合查询中的嵌套循环查询的底层思路是一样的，都是驱动表查询被驱动表，嵌套循环的伪代码如下：

可以用伪代码表示嵌套循环连接算法。

```
for each row R1 in the outer table
  for each row R2 in the inner table
    if R1 joins with R2
      return (R1, R2)
```

那我们为什么还要修改子查询为关联查询呢？因为带来的好处是，有关的访问路径，连接方法和连接顺序可能被有效使用，使得查询语句的层次尽可能的减少。在子查询中，In 语句外固定是驱动表，In 语句内固定是被驱动表，如果驱动表返回 1W 条记录，那被驱动表则会被查询 1W 次。而关联查询时，驱动表和被驱动表的关系是根据谁返回数据行的大小来动态决定的，如 A Join B，B 返回 10 行，A 返回 1W 行，则 SQL 底层执行时会将 B 作为驱动表，而 A 作为被驱动表，这样 B 执行一次查询，A 执行 10 次查询，反过来如果 A in(B)，则 A 执行一次查询，B 执行 1W 次查询。

代码示例 2：
and evl.[AutoModel] in (601,1473,1474,1993,1994,1995,2743,2750,2751,2752,3681,3682,3683,3684,5405,5819,5910,5911,10665,10666,10667,13444,13481,13482,13483,13484,13485,15377,15379,17807,17808,17809,17810,17811,17812)

修改方法：SQL 中关联 Product_Spec 表，
INNER JOIN dbo.Product_Spec prod_Spec ON prod_Spec.SpecId=evl.AutoModel
WHERE prod_Spec.SyearId=452 AND SpecState=(销售状态)

9.2.5、 关联子查询优化--子查询展开

非关联子查询的每次查询都依赖了驱动表的行数据内容，所以想修改为子查询展开方式不太好做，但我们可以从业务上分析，将相关的数据修改为不相关的数据集合，然后再跟不相关的数据集合进行关联，但跟不相关数据集合关联时，要带上条件。

代码示例：

```
and ((evl.[DrivenKiloms] >5000 and evl.[DrivenKiloms] <=20000) or evl.EvaluationID in (select EvaluationID from EvaluationAppending with(nolock) where DrivenKilometers>5000 and DrivenKilometers<20000 and RowStatus=0 and EvaluationAppendingID in (select MAX(EvaluationAppendingID) as EvaluationAppendingID from EvaluationAppending where RowStatus=0 group by EvaluationID)))) as
```

上面的逻辑含义是口碑的驱动公里数>5000 And <=20000，或者最后一条追加口碑的驱动公里数>5000 And <=20000，我们可以一次性把所有口碑的最后一条追加口碑并且是符合驱动公里数>5000 And <20000 的条件，保存到临时表或衍生数据表和 CTE 中，然后用这个临时表或 CTE 与口碑表进行关联查询，修改后的代码如下：

```
;WITH AppendEvalIDCTE AS(
  SELECT  EvaluationID
      FROM  ( SELECT  ROW_NUMBER() OVER ( PARTITION BY EvaluationID ORDER BY EvaluationAppendingID DESC ) AS RowNum ,
                EvaluationID ,DrivenKilometers
          FROM  EvaluationAppending with(nolock)
          WHERE  RowStatus = 0
        ) T
      WHERE  T.RowNum = 1 AND DrivenKilometers > 5000 AND DrivenKilometers <= 20000
)
```

关联代码：

```
INNER JOIN AppendEvalIDCTE evlAppend ON evl.EvaluationID = evlAppend.EvaluationID
  Where  evl.[DrivenKiloms] > 5000 AND evl.[DrivenKiloms] <= 20000
```

9.2.6、 关联子查询优化--谓词推入

谓词推入：当 SQL 语句中包含有不能合并的视图，并且视图有谓词过滤（也就是 where 过滤条件），CBO 会将 where 过滤条件推入视图中，这个就叫做谓词推入。谓词推入主要目的就是让 SQL Server 尽可能早的过滤掉无用的数据，从而提升查询性能。

当 SQL 语句中，OR 条件上面有一个为子查询，并且子查询上的表与源表不同，这个时候就可以用 union 代替 OR 或者你发现执行计划中的 filter 有 or 并且 or 后面跟上子查询(Exists)的时候就要注意，比如：

```
select * from test1 where owner='SCOTT' or object_id in(select object_id from test2 where owner='SCOTT');
```

修改为：

```
select * from test1 where owner='SCOTT'
union
select * from test1 where object_id in(select object_id from test2 where owner='SCOTT');
```

代码示例：

```
;WITH AppendEvalIDCTE AS(
   SELECT  EvaluationID
      FROM  ( SELECT  ROW_NUMBER() OVER ( PARTITION BY EvaluationID ORDER BY
```

```
EvaluationAppendingID DESC ) AS RowNum ,
                        EvaluationID ,DrivenKilometers
                FROM     EvaluationAppending with(nolock)
                WHERE    RowStatus = 0
              ) T
        WHERE   T.RowNum = 1 AND DrivenKilometers > 5000 AND DrivenKilometers <= 20000
 )
 , AllEvalIDCTE AS
 (
    SELECT HelpfulCount,LastAppend,Grade,AppendCount,EvaluationID FROM
    (
         SELECT                                          evl.HelpfulCount,evl.LastAppend,
evl.Grade ,evl.AppendCount,evl.EvaluationID FROM Evaluation evl
         INNER JOIN [Auto] at WITH ( NOLOCK ) ON at.AutoID = evl.AutoID
          WHERE  evl.[AutoSeries] =@id
                AND  evl.[DrivenKiloms] > 5000 AND evl.[DrivenKiloms] <= 20000
         UNION
          SELECT                                          evl.HelpfulCount,evl.LastAppend,
evl.Grade ,evl.AppendCount,evl.EvaluationID FROM Evaluation evl
          INNER JOIN [Auto] at WITH ( NOLOCK ) ON at.AutoID = evl.AutoID
        INNER JOIN AppendEvalIDCTE evlAppend ON evl.EvaluationID = evlAppend.EvaluationID
          WHERE  evl.[AutoSeries] =@id
                --不能再有了 AND  evl.[DrivenKiloms] > 5000 AND evl.[DrivenKiloms] <= 20000
    ) T
 )
```

9.2.7、 子查询优化—-Update 语句

update table_1 set score = score + 5 where uid in (select uid from table_2 where sid = 10);

其实 update 也可以用到 left join、inner join 来进行关联，可能执行效率更高，把上面的 sql 替换成 join 的方式如下：

update table_1 t1 inner join table_2 t2 on t1.uid = t2.uid set score = score + 5 where t2.sid = 10;

9.2.8、 子查询优化—-字段冗余

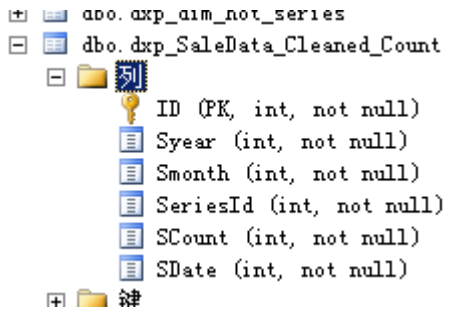
在 Evaluation 表新建一个 LastAppendingDrivenKilometers 字段
用于冗余存放 EvaluationAppending 表中当前口碑的最后一条追加数据的 DrivenKilometers 字段数据，代码就简洁多了。

9.2.9、 子查询优化—-物化视图（索引视图）

物化视图概念：对于涉及对大量的行进行复杂处理的非索引视图，为引用视图的每个查询动态生成结果集的开销会很大。这类视图包括聚集大量数据或联接许多行的视图。若经常在查询中引用这类视图，可通过在视图上创建唯一聚集索引来提高性能。在视图上创建唯一聚集索引时将执行该视图，并且结果集将存储在数据库中，就像带有聚集索引的表一样。

对于涉及对大量的行进行复杂处理的视图，由于结果集已经保存为一张带有聚集索引的表，因此无需重新计算，索引视图有明显的速度优势。

案例分析：




```
SELECT ps.BrandId ,SUM(tfw.SCount) SCount FROM [Replication].dbo.dxp_SaleData_Cleaned_Count tfw WITH(NOLOCK)INNER JOIN dbo.Product_Series ps WITH ( NOLOCK ) ON ps.SeriesId = tfw.SeriesId
                    WHERE ( tfw.Syear = @spFromYear AND tfw.Smonth >= @spFromMonth )
                        OR ( tfw.Syear = @spToYear AND tfw.Smonth < @spToMonth)
                        OR ( tfw.Syear > @spFromYear AND tfw.Syear < @spToYear)
                    GROUP BY ps.BrandId) s ON s.BrandId = sbf.BrandId
WHERE   1 = 1 AND sbf.CarType = 1 AND sbf.CountryId IN (1,2,3,4,5,6)
```

```
(8 行受影响)
表'dxp_SaleData_Cleaned_Count'。扫描计数1,逻辑读取378 次,物理读取0 次,预读0 次,lob 逻辑读取0 次,lob 物理读取0 次，lob 预读0 次。
表'Product_Series'。扫描计数1，逻辑读取10 次，物理读取0 次，预读0 次，lob 逻辑读取0 次，lob 物理读取0 次，lob 预读0 次。
 SQL Server 执行时间:
   CPU 时间= 15 毫秒，占用时间= 15 毫秒。
```

```
CREATE VIEW v_dxp_SaleData_Cleaned_Count WITH SCHEMABINDING AS
SELECT id,SeriesId,SCount,(Syear*100+Smonth))AS YearMonth
FROM dbo.dxp_SaleData_Cleaned_Count
CREATE  UNIQUE CLUSTERED INDEX v_dxp_SaleData_Cleaned_Count_Idx_ID ON v_dxp_SaleData_Cleaned_Count(id ASC);
 CREATE  NONCLUSTERED INDEX v_dxp_SaleData_Cleaned_Count_Idx_YearMonth ON v_dxp_SaleData_Cleaned_Count(YearMonth
```

```
ASC)INCLUDE(SeriesId,SCount);

;WITH CTE AS
(
SELECT ps.BrandId , SUM(tfw.SCount) SCount FROM
[Replication].dbo.v_dxp_SaleData_Cleaned_Count tfw WITH(NOLOCK)
INNER JOIN koubei.dbo.Product_Series ps WITH ( NOLOCK ) ON ps.SeriesId
= tfw.SeriesId
WHERE tfw.YearMonth>=201407 AND tfw.yearmonth<201511 GROUP BY ps.BrandId
)
SELECT  sbf.BrandId Id,sbf.BrandName
Name,sbf.PPH ,sbf.DefectiveSampleCount ,
       sbf.TroublefreeSamplesCount ,CTE.SCount
FROM   koubei.dbo.Stat_Brand_Fault sbf WITH ( NOLOCK ) INNER JOIN CTE  ON
CTE.BrandId = sbf.BrandId
 WHERE     1  =  1  AND  sbf.CarType  =  1  AND  sbf.CountryId  >=1  AND
sbf.CountryId<=6 -- (1,2,3,4,5,6)
```

```
表'v_dxp_SaleData_Cleaned_Count'。扫描计数1，逻辑读取34 次，物理读取0 次，预读0 次，lob 逻辑读取0
次，lob 物理读取0 次，lob 预读0 次。
表'Product_Series'。扫描计数1，逻辑读取10 次，物理读取0 次，预读0 次，lob 逻辑读取0 次，lob 物理读
取0 次，lob 预读0 次。

 SQL Server 执行时间:
   CPU 时间= 0 毫秒，占用时间= 5 毫秒。
```

9.3 、海量数据分页优化

9.3.1、 延迟索引

海量数据分页优化的 2 个重要思想就是：延迟索引和延迟关联。

举例：

select id,name from lx_com join userInfo between 5000000 and 5000010;

这个 SQL 中，它的工作原理是，他先取 500W 条，并且因为 name 字段不是主键，需要不断的回行到磁盘上去取，然后最后返回的是 500W 到 5000010 行数据，那么前 500W 行数据的 name，又要扔掉，不返给客户端，这样就形成极大的资源浪费。

修改为

With T as (select id from lx_com between 5000000 and 5000010)

select id,name from UserInfo join T on UserInfo.id=T.id

具体项目中的应用：

```
;WITH AllCTE AS( SELECT evl.EvaluationID,ROW_NUMBER() OVER(ORDER BY
evl.Grade desc,evl.AppendCount desc,evl.LastAppend desc)
AS RowNum FROM Evaluation evl WITH(NOLOCK)
WHERE  evl.[AutoSeries]=@id),evlCTE AS
(SELECT TOP 100 PERCENT EvaluationID,RowNum FROM AllCTE WHERE RowNum
BETWEEN @pagstart AND @pagend ORDER BY RowNum ASC)
```

9.3.2、 延迟关联

```
SELECT evlCTE.RowNum,evl.AppendCount,evl.EvaluationID,evl.AutoID,at.Brand AS
AutoBrand,evl.AutoModel,evl.AutoSeries
,evl.AutoOwner,at.[Level] AS AutoLevel,evl.CommentCount,evl.HelpfulCount
,evl.Created,evl.FeelingSummary ,at.BoughtDate AS
AutoBoughtDate,at.BoughtProvince AS AutoBoughtProvince
,at.BoughtCity AS AutoBoughtCity,
up.MemberId,up.NickName,up.IsAuthenticated,up.Gender,up.HeadImage,up.UserGrade,ps.Spec
Name
,ps.MinPrice,pss.SeriesName,pss.IsElectric
FROM evlCTE INNER JOIN Evaluation evl WITH(NOLOCK) ON evlCTE.EvaluationID =
evl.EvaluationID
INNER JOIN [Auto] at with(nolock) ON at.AutoID = evl.AutoID  INNER JOIN UserProxy up
with(nolock) ON up.UserID = evl.AutoOwner
INNER JOIN Product_Spec ps with(nolock) ON ps.SpecId=evl.AutoModel
INNER JOIN Product_Series pss with(nolock) ON pss.SeriesId=evl.AutoSeries ORDER BY
evlCTE.RowNum ASC;
```

9.3.3、 求总行数的性能优化

1、 连接消除

在求总行数是，凡是仅仅是因为显示数据列需要关联的表，就可以不用再关联了。

2、 排序消除

排序是需要很大消耗的，在求总行数时，不需要再用到排序。

3、 与分页数据查询放在同一个查询中执行，这样可以减少网络连接与传输

第一笔数据就是分页数据，比如：第 10 条到第 20 条数据，另外一笔数据是当前查询的总记录数，那么我们可以用一个”;”分号进行 SQL 来实现一次查询取出 2 笔数据，然后在 C#的 DataSet 对象中分别从 Table[0]和 Table[1]中取出这 2 笔不同的数据。

简化后的代码如下：

```
SELECT COUNT(*)AS TotalCount
 FROM Evaluation evl WITH(NOLOCK) WHERE  evl.RowStatus=0 AND evl.[AutoSeries]=@id and evl.[EditedFeeling]=1 and evl.[Grade]>-10
```

9.4 、With As 与性能优化

对于 SELECT 查询语句来说,通常情况下，为了使 T-SQL 代码更加简洁和可读，在一个查询中引用另外的结果集都是通过视图而不是子查询来进行分解的.但是，视图是作为系统对象存在数据库中，那对于结果集仅仅需要在存储过程或是用户自定义函数中使用一次的时候，使用视图就显得有些奢侈了.

公用表表达式（Common Table Expression)是 SQL SERVER 2005 版本之后引入的一个特性.CTE 可以看作是一个临时的结果集，可以在接下来的一个 SELECT,INSERT,UPDATE,DELETE,MERGE 语句中被多次引用。使用公用表达式可以让语句更加清晰简练.

除此之外，根据微软对 CTE 好处的描述，可以归结为四点:

- 可以定义递归公用表表达式(CTE)
- 当不需要将结果集作为视图被多个地方引用时，CTE 可以使其更加简洁
- GROUP BY 语句可以直接作用于子查询所得的标量列
- 可以在一个语句中多次引用公用表表达式(CTE)
- 注意：如果 with as 短语被调用了 2 次以上，CBO 会自动将 with as 短语的数据放入一个临时表。

在 MSDN 中的原型：

```
WITH expression_name [ ( column_name [,...n] ) ]

AS

( CTE_query_definition )
```

代码示例：

```
WITH CTE_Test
AS
(
SELECT *
FROM HumanResources.Employee
)

SELECT * FROM CTE_Test
```