

# 性能调优实习报告

<https://github.com/AOZMH/TPC-H-Optimization-Project>

1700017815 张旻昊 1700017802 耿思博 1700017828 胡时京

## 【目录】

### 一、运行环境及准备工作

运行系统配置、数据库软件等介绍

DBGEN 编译、数据生成与导入及实验准备工作

### 二、各查询分析及优化

首先对每个查询在未建立索引的情况下给出基准性表现（含运行时间、IO 统计）

其次利用索引、表分区、物化视图等方式优化查询、对比性能并进行分析和进一步实验

### 三、总体性能比较

建立所有索引并调整相应数据库参数后，在所有查询上统一测评

### 四、总结

## 一、运行环境及准备工作

### 1、运行环境

CPU	Intel Core i7-7700HQ @ 2.80GHz
内存	16GB RAM
磁盘	1TB 机械硬盘
操作系统	Windows 10 1903
数据库	Microsoft SQL Server 2019 15.2002.4709.1

### 2、准备工作

#### 2.1、数据导入

为生成 SQL Server 中可用的数据及查询，修改 makefile.suite 文件设置 DATABASE = SQLSERVER, MACHINE = WIN32, 在 Visual Studio 中生成 DBGEN 和 QGEN 可执行文件，之后运行 dbgen -vf -s 1 生成 1G 数据，运行 qgen 将模板 sql 语句转为可以在 SQL Server 中运行的 SQL 语句。

随后，运行 create.sql 创建 TPC-H 数据库并创建相关表格，注意，主码约束在 create.sql 中以 PRIMARY KEY 的方式已经建立。随后，运行 load\_data.py 通过 pymssql 包导入之前生成的 8 个.tbl 数据。至此，所有数据都被成功导入。

#### 2.2、实验前的准备

在进行实验前，运行 `set statistics IO on` 和 `set statistics time on`，显示时间和访存次数的统计信息，用以评估查询性能。此外，在每次查询前，运行如下操作清空数据库缓存；这是由于执行过一条语句后，相应内容会从磁盘读入缓存（如内存），这样之后重复执行会快很多，为评估我们做的优化本身的作用（而非缓存的作用），运行此语句以规范化。

```
dbcc dropcleanbuffers
dbcc freeproccache
```

## 二、各查询分析及优化

### 1、Query 1

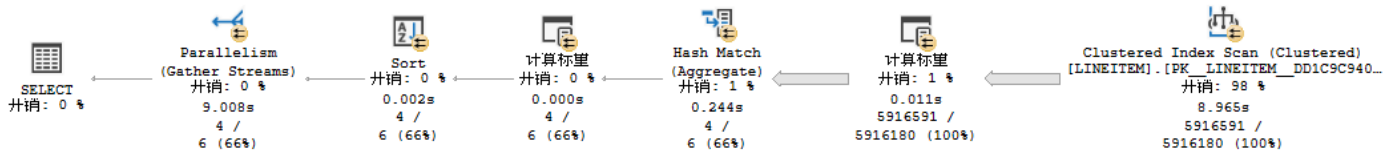
#### 1.1、基准结果

首先，在没有任何索引的基础上运行查询，检查运行时间、执行计划及访存信息，如下。

SQL Server 执行时间：  
CPU 时间 = 2278 毫秒，占用时间 = 9138 毫秒。  
SQL Server 分析和编译时间：  
CPU 时间 = 0 毫秒，占用时间 = 0 毫秒。

(4 行受影响)

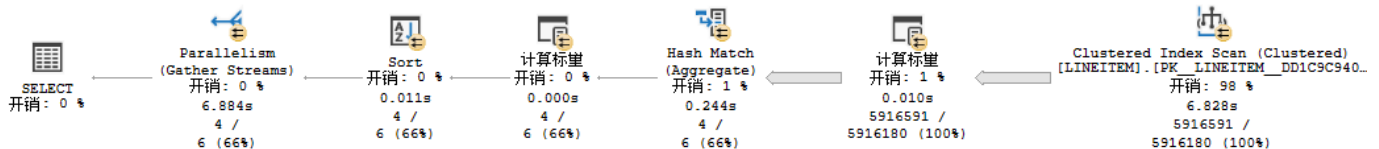
表"LINEITEM"。扫描计数 9，逻辑读取次数 104799，物理读取次数 2，页面服务器读取次数 0，预读读取次数 104013，页面服务器预读读取次数 0  
表"Worktable"。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑



经验证，查询答案正确。

#### 1.2、索引优化分析

首先，由于我们的查询不存在并发的情形，可以接受read uncommitted的隔离性级别，因此先设置该隔离性级别，再次执行，所得结果如下。



(4 行受影响)

表"LINEITEM"。扫描计数 9，逻辑读取次数 104029，物理读取次数 0，页面服务器读取次数 0，预读读取次数 103945，页面服务器预读读取次数 0  
表"Worktable"。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑

可见，时间由9s缩短至7s，但是读取次数几乎没有变化（事实上每次执行，逻辑读取次数都略有不同，存在随机因素），这符合预期——修改隔离性级别并不能减少读取次数，它仅能减小并发控制时带来的额外开销，因此总查询时间缩短。

同时，从执行计划中可以看出，全表扫描占用了绝大多数的时间，且该扫描是在主码的聚集索引上进行的，因此时间开销很大，我们需要建立索引以减小访存带来的时间开销。Q1仅针对于LINEITEM一张表，根据L\_SHIPDATE列的值选择一些行，并利用group by展示一些信息。显然，我们应该针对L\_SHIPDATE建立非聚集索引（由于已有主码，所以已有聚簇索引，只能建立非聚集索引）；另外，为了让整个查询只需在索引表上执行，可以建立覆盖索引，即将SELECT和GROUP BY、ORDER BY的位域include入索引中。如下所示。

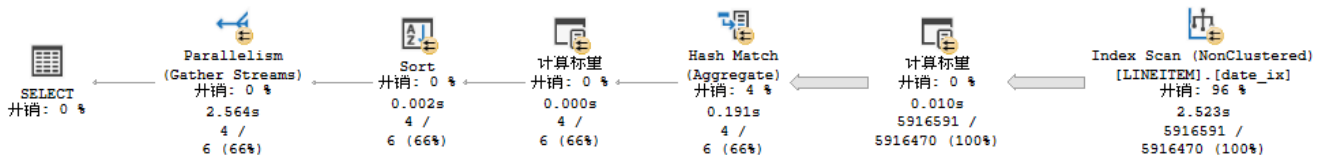
```
create nonclustered index date_ix on LINEITEM(L_SHIPDATE)
include (L_QUANTITY, L_EXTENDEDPRICE, L_DISCOUNT, L_TAX,
L_RETURNFLAG, L_LINESTATUS);
```

建立索引后再次执行Q1，结果如下。

SQL Server 执行时间：  
CPU 时间 = 2014 毫秒，占用时间 = 2878 毫秒。  
SQL Server 分析和编译时间：  
CPU 时间 = 0 毫秒，占用时间 = 0 毫秒。

(4 行受影响)

表"LINEITEM"。扫描计数 9，逻辑读取次数 41785，物理读取次数 0，页面服务器读取次数 0，预读读取次数 40934，页面服务器预读读取次数 0  
表"Worktable"。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑



可见，时间显著减少，由6.8s进一步缩减至2.5s。为什么会有如此优化？观察statistics IO即可发现，由于索引的建立，逻辑读取次数显著减小，同时执行计划中的全表扫描也变成了索引扫描，由于索引表远小于LINEITEM全表，扫描更快，这使得总体性能显著提升。

基于Q1的实践经验，将隔离性级别设置为read uncommitted有助于性能提升、反应索引的作用，因此在此之后的实验中全部将优化后查询，设置为read uncommitted级别。

### 1.3、表分区优化分析

接下来考虑表分区带进来的优化效果。由于where子句中仅依靠SHIPDATE进行约束，我们应首先考虑基于SHIPDATE对LINEITEM表进行分区。为进行分组，首先通过add filegroup建立5个文件组group1~group5，再通过add file将文件组指定到相应磁盘物理文件中，以下面group1的添加为例，可以新建5个文件组，用以让表分到5个独立的区域内。

```
alter database TPCH add filegroup group1
ALTER DATABASE TPCH ADD
FILE (NAME=N' tpch_group1', FILENAME=N'D:\xxx\tpch_group1.ndf', SIZE=3MB,
MAXSIZE=UNLIMITED, FILEGROWTH=5MB) TO FILEGROUP group1
```

此后，建立分区函数和分区方案，最后再将分区方案应用到LINEITEM表即可。根据上述分析，分区函数应该以一个date型数据为键（表中的SHIPDATE），且应设置4个分隔值，这样可以将表分入5个区域。为了尽可能提高性能，应该使得让分区均与，换言之，我们应该尽可能找到LINEITEM关于SHIPDATE的五等分点。为此，进行测试，不断更改日期进行如下查询（其中6001215为LINEITEM总长度），找到使得查询结果为0.2、0.4、0.6、0.8的日期，即为五等分点。

```
select count(*)/6001215.0
from LINEITEM
where L_SHIPDATE <= '1997-06-15';
```

此后，即可利用下面的方式创建分区函数和对应的分区方案。

```
CREATE PARTITION FUNCTION tpch_partition_qldate( DATE )
AS RANGE RIGHT
FOR VALUES ( '1993-07-01', '1994-10-24', '1996-03-01', '1997-06-15' );

CREATE PARTITION SCHEME tpch_partition_scheme_qldate
AS PARTITION tpch_partition_qldate
TO (GROUP1, GROUP2, GROUP3, GROUP4, GROUP5 );
```

最后，新建LINEITEM\_PAR表格，定义与LINEITEM相同，只不过最后加上字句 on tpch\_partition\_scheme\_qldate( L\_SHIPDATE ) 应用分区方案进行分区。最后，执行 insert into LINEITEM\_PAR select \* from LINEITEM将所有数据同步到新表中。此时之前指定的分区文件如下，可见已经容纳了相应量的数据，且总和约与LINEITEM总大小相同，这说明我们成功地将LINEITEM进行分区存储。

tpch_group1.ndf	228,352 KB
tpch_group2.ndf	259,072 KB
tpch_group3.ndf	1,175,552...
tpch_group4.ndf	248,832 KB
tpch_group5.ndf	300,032 KB

此时，在LINEITEM\_PAR上执行Q1查询，性能如下。可见，总体执行时间较baseline有一定提升，但效果很不明显。进一步分析，可以发现where子句对SHIPDATE的限制很松，这导致95%的数据一定会被select到，因此表分区意义实际上不大——反正各个分区的内容也都会被select出来进入聚集函数中。因此，对于此查询，由于索引让表扫描本身变快，而非缩小搜索范围，其效果更好。

SQL Server 执行时间：  
CPU 时间 = 2298 毫秒，占用时间 = 7536 毫秒。  
SQL Server 分析和编译时间：  
CPU 时间 = 0 毫秒，占用时间 = 0 毫秒。

(4 行受影响)  
表“LINEITEM\_PAR”。扫描计数 11，逻辑读取次数 103543，物理读取次数 0，页面服务器读取次数 0，预读读取次数 95139，页面服务器预读读取次数 0，表“Worktable”。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑读取次数 0。

#### 1.4、物化视图优化分析

SQL Server中的物化视图被称为indexed views，它通过建立一个符合一定条件的视图后，通过在其上建立唯一性聚集索引，让DBMS自动将整个视图物理地存储下来，这样实现物化视图的思想。为了在对表动态更改的条件下维护物化视图，SQL Server要求视图不可有非确定性、非精确、多个聚集函数组合运算（如AVG或SUM/COUNT）的列，而Q1中AVG列就不符合这一条件，因此在创建物化视图时，需要在视图中分别记录SUM和COUNT信息，在查询时再进行后处理。这样虽然无法让物化视图“一步到位”直接存储查询结果，但仍然省掉了很多的表运算步骤（包括group by、where等），可提高效率。具体地，按照SQL Server规定，先set如下参数以创建物化视图。

```
SET NUMERIC_ROUNDABORT OFF;  
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT, QUOTED_IDENTIFIER, ANSI_NULLS ON;
```

其次，创建常规视图dbo.query1对应于Q1的相关结果，基于上述分析，我们将用于计算AVG的sum和count分开存储于视图中，如下所示。

```
create view dbo.query1  
with schemabinding  
as  
    SELECT L_RETURNFLAG, L_LINESTATUS,  
           SUM(L_EXTENDEDPRI) AS SUM_BASE_PRICE, SUM(L_EXTENDEDPRI*(1-L_DISCOUNT)) AS SUM_DISC_PRICE,  
           SUM(L_EXTENDEDPRI*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE,  
           SUM(L_QUANTITY) AS SUM_QTY, COUNT_BIG(L_QUANTITY) AS CT_QTY,  
           SUM(L_EXTENDEDPRI) AS SUM_PRICE, COUNT_BIG(L_EXTENDEDPRI) AS CT_PRICE,  
           SUM(L_DISCOUNT) AS SUM_DISC, COUNT_BIG(L_DISCOUNT) AS CT_DISC,  
           COUNT_BIG(*) AS COUNT_ORDER  
    FROM dbo.LINEITEM  
    WHERE L_SHIPDATE <= dateadd(dd, -90, CONVERT(DATETIME, '1998-12-01', 102))  
    GROUP BY L_RETURNFLAG, L_LINESTATUS  
;
```

最后，在query1上建立唯一性聚簇索引ix1，以group by的两个键为key即可，如下所示。此时，dbo.query1视图就被物理的存储与磁盘上了，此后查询相关内容可直接使用其信息。

```
create unique clustered index ix1 on dbo.query1 ( L_RETURNFLAG, L_LINESTATUS );
```

首先检验物化视图的正确性，做如下查询，相当于“包装”了query1视图，加上后处理，得到与Q1相同的语义。运行这一查询发现，结果与Q1结果（q1.out）相同，可见物化视图的语义正确。

```
select L_RETURNFLAG, L_LINESTATUS, SUM_QTY, SUM_BASE_PRICE, SUM_DISC_PRICE, SUM_CHARGE,
       SUM_QTY/CT_QTY AS AVG_QTY, SUM_PRICE/CT_PRICE AS AVG_PRICE, SUM_DISC/CT_DISC AS AVG_DISC, COUNT_ORDER
from dbo.query1 ORDER BY L_RETURNFLAG, L_LINESTATUS;
```

观察此查询的性能表现，如下。可见，总体占用时间非常显著的减少（9s->0.1s），这比索引的性能还要好。其主要原因也可以从IO结果发现——建立物化视图后，只需要在query1表上进行查询，无需访问LINEITEM，而query1实际上是group by之后的结果，因此只有4行，体量是远小于LINEITEM及之前LINEITEM上的覆盖索引表的，因此在其上查询自然很快。这也反应了物化视图的优越性——在可以将查询的绝大部分放入物化视图时，它对查询性能的提升是巨大的，因为绝大多数运算被避免了。

```
SQL Server 执行时间:
CPU 时间 = 0 毫秒, 占用时间 = 124 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(4 行受影响)  
表“query1”。扫描计数 1, 逻辑读取次数 2, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 逻辑读取次数 0

建立此物化视图后，再**查询原查询Q1**（上面只是运行了包装查询检验正确性、有效性，为了统一比较还是应该基于原查询跑一遍结果），结果如下。可见，总运行时间约为0.14s, 同样显著低于baseline，达到了非常好的性能。IO结果与上面的结果相似，这说明建立物化视图后，DBMS可以自动在查询中识别可以使用物化视图的部分，进行优化，提高性能。

```
SQL Server 执行时间:
CPU 时间 = 0 毫秒, 占用时间 = 140 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(4 行受影响)  
表“query1”。扫描计数 1, 逻辑读取次数 2, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 逻辑读取次数 0

进一步查看执行计划，这验证了上面的断言：所有查询只需要在物化视图的聚簇索引上完成并做一些后处理即可。这证明的了物化视图的优点——它是“对用户屏蔽的”，即建立好合适的物化视图后，用户无需更改原查询，系统即可自动将物化视图在可能时放入查询中，优化性能。



然而，物化视图也拥有一定的问题：如果它与查询极为接近（例如上面的例子），则它可以对相应的查询有极大的优化，但对于其他大多数查询则难以用到；如果要让它用到更多的查询中，则优化幅度就没有那么明显，这需要用户权衡；另外，物化视图同样需要额外的开销存储视图，与索引相同，使得其在空间开销上不如表分区。总之，上述三种优化方式（索引、表分区、物化视图）在Q1上各有优劣，但均起到了一定的优化效果，之后会在更多query上进一步探讨。

## 2、Query 2

### 2.1、基准结果

不建立索引，在read committed（默认）下直接执行查询，统计信息如下。经验证，结果正确。



```
SQL Server 执行时间:
CPU 时间 = 251 毫秒, 占用时间 = 2701 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(100 行受影响)

表“SUPPLIER”。扫描计数 18, 逻辑读取次数 1202, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 193, 页面服务器预读读取次数 0  
表“PARTSUPP”。扫描计数 469, 逻辑读取次数 25742, 物理读取次数 2, 页面服务器读取次数 0, 预读读取次数 23679, 页面服务器预读读取次数 0  
表“NATION”。扫描计数 15, 逻辑读取次数 8, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑  
表“REGION”。扫描计数 11, 逻辑读取次数 8, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑  
表“PART”。扫描计数 9, 逻辑读取次数 3860, 物理读取次数 3, 页面服务器读取次数 0, 预读读取次数 3650, 页面服务器预读读取次数 0, LOB 逻辑  
表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑  
表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑

## 2.2、索引优化分析

【优化1】Q2涉及5表连接，此时优化对最大的表的查询就是非常重要的了。经过实验，PARTSUPP表是最大的表（有800000行），而该表的主码为<PARTKEY, SUPPKEY>。这在查询时会造成性能降低，原因在于在外层where语句中，SUPPKEY需要分别利用两个KEY和PART、SUPPLIER表连接，此时两个等于条件没法各自利用PARTSUPP上的索引，因此只能在聚簇索引上全表扫描。对此，我们只需在两个SUPPKEY上建立索引，并将PARTKEY等其他查询涉及的列 include入覆盖索引即可，如下所示。

```
create nonclustered index ix1 on PARTSUPP (PS_PARTKEY)
include (PS_SUPPLYCOST, PS_SUPPKEY);
```

此时再执行Q2，结果如下。可见执行时间显著减少，访存也明显减少；这是因为在外层where中SUPPKEY与SUPPLIER连接时使用了索引，迅速减少了搜索范围。

```
SQL Server 执行时间:
CPU 时间 = 156 毫秒, 占用时间 = 686 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(100 行受影响)

表“SUPPLIER”。扫描计数 2, 逻辑读取次数 402, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 201, 页面服务器预读读取次数 0  
表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑  
表“PARTSUPP”。扫描计数 461, 逻辑读取次数 3675, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 2143, 页面服务器预读读取次数 0  
表“PART”。扫描计数 1, 逻辑读取次数 3674, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 3674, 页面服务器预读读取次数 0, LOB 逻辑  
表“NATION”。扫描计数 2, 逻辑读取次数 4, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑  
表“REGION”。扫描计数 1, 逻辑读取次数 5, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑  
表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑

【优化2】更进一步地，我们还可以建立更多索引。然而，其他需要添加索引的列均为KEY列，而PART、SUPPLIER等表的KEY都建立了聚簇索引，因此实际上再建立非聚集索引意义不大，作为实验，建立如下索引，再执行查询，结果如下。

```
create nonclustered index ix1 on SUPPLIER (S_SUPPKEY) include (S_NATIONKEY,
S_ACCTBAL, S_NAME, S_ADDRESS, S_PHONE, S_COMMENT);
create nonclustered index ix1 on NATION (N_NATIONKEY) include (N_NAME);
```

```
SQL Server 执行时间:
CPU 时间 = 62 毫秒, 占用时间 = 669 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(100 行受影响)

表“SUPPLIER”。扫描计数 2, 逻辑读取次数 394, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, 表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, 表“PARTSUPP”。扫描计数 461, 逻辑读取次数 3675, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 2143, 页面服务器预读读取次数 0, 表“PART”。扫描计数 1, 逻辑读取次数 3674, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 3674, 页面服务器预读读取次数 0, 表“NATION”。扫描计数 2, 逻辑读取次数 4, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 表“REGION”。扫描计数 1, 逻辑读取次数 52, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0,

可见, 运行时间略有减少, 读盘次数也有小幅度减小, 但建立索引本身也有时间、空间代价, 这种小幅度优化实际上意义不大。

【优化3】除此之外, 不难发现, 外层where中有P\_SIZE=15的条件语句; 因此, 实际上PART表有两个约束, 一个基于PARTKEY, 另一个基于SIZE; PARTKEY的索引为聚簇索引, 扫描较慢, 另外, SIZE的约束范围较小、判别较快, 因此我们不妨再SIZE上建立非聚集索引, 如下。

```
create nonclustered index ix1 on PART (P_SIZE)
include (P_MFGR, P_TYPE);
```

此后再执行Q2, 结果如下。可见, PART表的读取次数显著减少(3674->36), 这正是由于建立了合适的索引, 在查询早期就利用ix1快速缩小了范围, 最终执行时间也显著减少。

```
SQL Server 执行时间:
CPU 时间 = 47 毫秒, 占用时间 = 244 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(100 行受影响)

表“SUPPLIER”。扫描计数 2, 逻辑读取次数 394, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 197, 页面服务器预读读取次数 0, 表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 表“PARTSUPP”。扫描计数 461, 逻辑读取次数 3675, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 2296, 页面服务器预读读取次数 0, 表“PART”。扫描计数 1, 逻辑读取次数 36, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 37, 页面服务器预读读取次数 0, Lob 表“NATION”。扫描计数 2, 逻辑读取次数 4, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 表“REGION”。扫描计数 1, 逻辑读取次数 52, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob

总之, 本查询的执行时间从2700ms优化到240ms, 效果明显。同时, 从中也不难发现: ①不是加索引就好, 有时候加索引开销大, 同时也没有什么性能提升; ②在多个列均可加索引时, 应该优先选择条件约束高、索引key占用空间小、条件判定速度快的(例如int的=判定), 这样访问索引表时效率高。

### 2.3、物化视图优化分析

Q2比较复杂, 具有一个相关子查询, 所以建立物化视图是一件较为困难的事情。首先, Q2无法像Q1一样直接将整个查询结果直接存入物化视图中, 大幅度提高性能——由于子查询是相关子查询, SQL Server不允许在这样的视图上建立索引, 我们应想办法将子查询部分与其他where部分查分处理。其次, Q2的答案行数较多, 仅是最后通过TOP 100选取了100行, 由于物化视图无法加入TOP子句(这个很容易理解, 因为物化视图需要动态维护, 如果它有TOP K, 那插入或删除后就无法维护了, 因此需要保存所有select的结果), 不论如何处理物化视图中均会有大量的数据行。相比起Q1答案仅有4行, 大量答案行使物化视图的存储、访问、计算开销变大, 这同样使得像Q1一样“一步到位”地添加物化视图变得不合理。我们需要尝试新的优化方法。

根据上述分析, 首先应该尝试将子查询分离。由于子查询与外层where字句中的P\_PARTKEY列相关, 所以无法通过一个视图直接计算出子查询的结果。因此, 采用多加列的方法: 将与P\_PARTKEY关联的PS\_PARTKEY列连同select的列一同记录在视图中, 这样在执行Q2时通过扫描此物化视图, 将P\_PARTKEY与PS\_PARTKEY匹配, 即可得到select的内容。因此, 建立的第一个物化视图如下。

```

drop view if exists dbo.query2_1;
create view dbo.query2_1
with schemabinding as
    SELECT PS_PARTKEY, PS_SUPPLYCOST FROM dbo.PARTSUPP, dbo.SUPPLIER, dbo.NATION, dbo.REGION
    WHERE S_SUPPKEY = PS_SUPPKEY AND S_NATIONKEY = N_NATIONKEY
    AND N_REGIONKEY = R_REGIONKEY AND R_NAME = 'EUROPE';

create unique clustered index ix1
on dbo.query2_1( PS_PARTKEY, PS_SUPPLYCOST )

```

添加此物化视图后执行Q2，结果如下。

```

SQL Server 执行时间:
CPU 时间 = 109 毫秒, 占用时间 = 2104 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。

```

表“SUPPLIER”。扫描计数 1, 逻辑读取次数 202, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 193, 页面服务器预读读取次数 0  
 表“query2\_1”。扫描计数 1, 逻辑读取次数 436, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, I  
 表“PART”。扫描计数 1, 逻辑读取次数 3675, 物理读取次数 3, 页面服务器读取次数 0, 预读读取次数 3650, 页面服务器预读读取次数 0, I  
 表“NATION”。扫描计数 1, 逻辑读取次数 2, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB  
 表“REGION”。扫描计数 1, 逻辑读取次数 2, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB  
 表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, L  
 表“PARTSUPP”。扫描计数 460, 逻辑读取次数 3723, 物理读取次数 9, 页面服务器读取次数 0, 预读读取次数 4192, 页面服务器预读读取次

可见，总占用时间相较baseline有一定减少，IO方面变换更加明显：对PARTSUPP的访问次数显著变低，由于PARTSUPP最大，因此执行时间有了优化。然而，毕竟这一物化视图只解决了一部分问题，子查询外的优化还没有涉及，因此继续尝试添加其他物化视图。

注意到子查询外的where语句实际上是多个表的连接加上一些数值约束，因此可以考虑将这些连接的过程在物化视图中完成；为了使这个视图也引用到最终的查询中，将Q2中select的列、与子查询连接所需要的PS\_SUPPLYCOST及子查询中涉及的列P\_PARTKEY都放进select中，得到新视图dbo.query2\_2。此后建立聚簇索引ix1，即可实现物化视图。

```

create view dbo.query2_2
with schemabinding as
    SELECT S_SUPPKEY, S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY, P_MFGR, S_ADDRESS, S_PHONE,
    S_COMMENT, PS_SUPPLYCOST
    FROM dbo.PART, dbo.SUPPLIER, dbo.PARTSUPP, dbo.NATION, dbo.REGION
    WHERE P_PARTKEY = PS_PARTKEY AND S_SUPPKEY = PS_SUPPKEY AND P_SIZE = 15 AND
    P_TYPE LIKE '%%BRASS' AND S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY AND
    R_NAME = 'EUROPE';

create unique clustered index ix1
on dbo.query2_2( P_PARTKEY, S_SUPPKEY )

```

此时执行原查询，性能表现如下。可见，在查询中query2\_2并没有被使用，结果也与上面几乎相同。

```

SQL Server 执行时间:
CPU 时间 = 78 毫秒, 占用时间 = 2200 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。

```

(100 行受影响)

表“SUPPLIER”。扫描计数 1, 逻辑读取次数 202, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 193, 页面服务器预读读取次数 0  
 表“query2\_1”。扫描计数 1, 逻辑读取次数 436, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, I  
 表“PART”。扫描计数 1, 逻辑读取次数 3675, 物理读取次数 3, 页面服务器读取次数 0, 预读读取次数 3650, 页面服务器预读读取次数 0, I  
 表“NATION”。扫描计数 1, 逻辑读取次数 2, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB  
 表“REGION”。扫描计数 1, 逻辑读取次数 2, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB  
 表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, L  
 表“PARTSUPP”。扫描计数 460, 逻辑读取次数 3723, 物理读取次数 9, 页面服务器读取次数 0, 预读读取次数 4192, 页面服务器预读读取次



为了探究这一问题，我又尝试了几种视图建立方式，例如在where中不添加数值上的约束，仅完成表连接；又如在建立聚簇索引时以不同位域作为key等。但不论如何，SQL Server始终无法识别这一物化视图并在查询时使用它。在查询了一些Microsoft SQL Server文档及StackOverflow论坛帖子后，我发现有人指出SQL Server通常不会在一个查询中使用多个“有交叠”的物化视图。具体地，对于本问题，query2\_1和query2\_2并非简单地加入查询中，相反，需要将原查询进行改写才可以合并这两个物化视图（因为子查询是相关的，所以需要相关的判断，即P\_PARTKEY=PS\_PARTKEY转换到两个物化视图的相应列相等判断），此时数据库引擎可能无法识别，进而无法将query2\_2同时融入查询中。因此，就目前的实验而言，物化视图对本查询有一定优化，但效果不如索引明显。从中我们也可以有一点收获：对于嵌套关系较复杂（多层嵌套、相关子查询等）的query，往往物化视图难以起到非常好的效果，因为很难从这些复杂的查询中“独立”出来一部分作为物化视图。在这些情况中，我们应优先考虑索引策略的应用。

### 3、Query 3

#### 3.1、基准结果

在不建立任何索引的条件下运行查询，基准结果如下。

```
SQL Server 执行时间:
CPU 时间 = 1921 毫秒, 占用时间 = 13708 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(10 行受影响)  
表"ORDERS"。扫描计数 9, 逻辑读取次数 22239, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 22010, 页面服务器预读读取次数 0  
表"LINEITEM"。扫描计数 9, 逻辑读取次数 104804, 物理读取次数 2, 页面服务器读取次数 0, 预读读取次数 104013, 页面服务器预读读取次数 0  
表"CUSTOMER"。扫描计数 0, 逻辑读取次数 1081, 物理读取次数 239, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0,  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 142, 页面服务器预读读取次数 0, LOb  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOb  
经检验，查询结果正确无误。

#### 3.2、索引优化分析

Q3将CUSTOMER、ORDERS、LINEITEM三个表连接，并在where子句中对三个表均加以非主码上的条件限制，最后得到查询结果。因此，我们可以在约束对应的列上添加索引。

与之前的优化类似，在L\_SHIPDATE、O\_ORDERDATE、C\_MKTSEGMENT上添加索引，同时包含查询可能涉及的其他位域，如下所示。

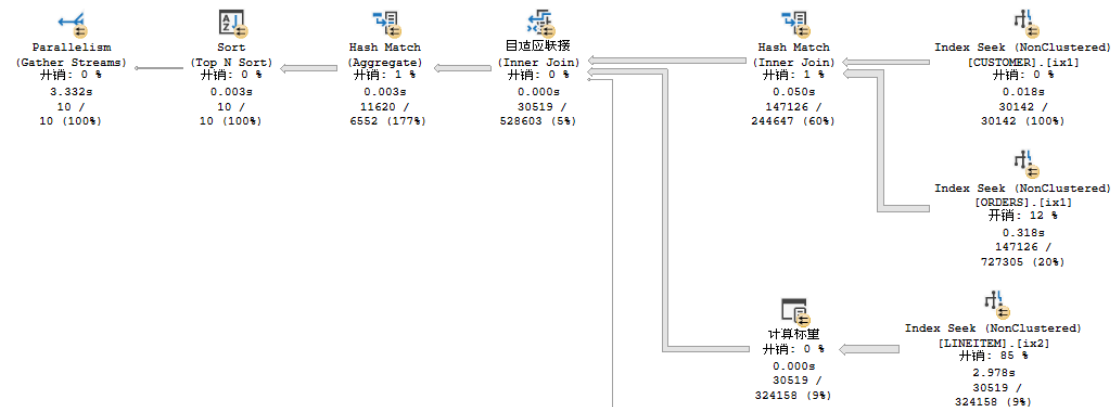
```
create nonclustered index ix2 on LINEITEM(L_SHIPDATE) include (L_EXTENDEDPRICE, L_DISCOUNT);
create nonclustered index ix1 on ORDERS(O_ORDERDATE) include (O_CUSTKEY, O_SHIPPRIORITY);
create nonclustered index ix1 on CUSTOMER(C_MKTSEGMENT);
```

此时执行Q3，统计结果如下。可以发现执行时间显著减少，仅用了原来的1/4左右；访存次数的减少则更明显，仅约为原来的1/10左右。

```
SQL Server 执行时间:
CPU 时间 = 359 毫秒, 占用时间 = 3427 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(10 行受影响)  
表"LINEITEM"。扫描计数 9, 逻辑读取次数 14227, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 14093, 页面服务器预读读取次数 0  
表"ORDERS"。扫描计数 9, 逻辑读取次数 1991, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 1689, 页面服务器预读读取次数 0, LOb  
表"CUSTOMER"。扫描计数 9, 逻辑读取次数 307, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 31, 页面服务器预读读取次数 0, LOb  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOb  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOb

进一步分析原因,查看执行计划如下。可以发现,占用时间最多的右侧三个表扫描,均使用了Index Seek,没有使用主码上的聚簇索引;而且我们使用了覆盖索引,因此所有搜索均可在索引表上完成。这样一来,由于索引表小于原表,访存次数显著减少,运行时间也相应缩短。



### 3.3、表分区优化分析

#### 【方案1】

由于查询中有对三个表的非主码列的约束,我们可以考虑分别在这些列上将对应的表分区。具体的,新建5个文件组(新建文件组的代码已经在Q1中介绍过,如add\_group.sql所示,之后的表分区实验均在这些文件组上进行),依次建立针对当前数据的分区函数、分区方案,如下所示。

```
CREATE PARTITION FUNCTION tpch_partition_q3date( DATE )
AS RANGE RIGHT
FOR VALUES( '1993-05-01', '1994-09-01', '1995-12-31', '1997-04-15' );

CREATE PARTITION SCHEME tpch_partition_scheme_q3date
AS PARTITION tpch_partition_q3date
TO (GROUP1, GROUP2, GROUP3, GROUP4, GROUP5 );

drop partition function tpch_partition_q3char;
CREATE PARTITION FUNCTION tpch_partition_q3char ( char(10) ) AS
RANGE LEFT FOR VALUES (0,1,2,3)

drop partition scheme tpch_partition_scheme_q3char
CREATE PARTITION SCHEME tpch_partition_scheme_q3char
AS PARTITION tpch_partition_q3char TO (GROUP1, GROUP2, GROUP3, GROUP4, GROUP5 );
```

这里q3date应用于ORDERS和LINEITEM表,基于date数据类型分区,q3char应用于CUSTOMERS表,利用char(10)字符串进行分区。

与Q1中方法相同,由于SQL Server不支持在已经存在的表上重新分区,新建CUSTOMER\_PAR、ORDERS\_PAR、LINEITEM\_PAR表,并在最后添加: on tpch\_partition\_scheme\_q3char( C\_MKTSEGMENT )、 on tpch\_partition\_scheme\_q3date( O\_ORDERDATE )、 on tpch\_partition\_scheme\_q3date( L\_SHIPDATE )使得分区方案应用到表上。注意,此处添加的新表与原表有一点不同,他们都没有设定主码,这是因为SQL Server为所有设定主码的表添加聚簇索引,同时还规定表分区的键必须包含于聚簇索引的码中,因此若此时分区键并非主码,则无法添加相应分区

方案。由于主码约束对于我们的静态查询没有影响（没有增删数据操作），为了评估效果，这里删除了主码约束。

最后，再通过 insert into A select \* from B 将原表内容重复插入到新表即可，这样即得到了数据相同但分布在GROUP1~GROUP5的表格。

此时，再运行Q3，结果如下。一方面，运行时间较baseline显著降低到约0.4，加速效果明显；另一方面，访存次数并没有明显减少，这说明后台执行时虽然并发多次读取了各个文件组，但由于并行性，差不多的访存次数最终执行时间反而更短，这体现了表分区的作用。

```
SQL Server 执行时间:
CPU 时间 = 624 毫秒, 占用时间 = 5770 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(10 行受影响)

表"LINEITEM\_PAR"。扫描计数 11, 逻辑读取次数 64366, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 64324, 页面服务器预读读取次数 0  
表"ORDERS\_PAR"。扫描计数 11, 逻辑读取次数 13300, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 13293, 页面服务器预读读取次数 0, I  
表"CUSTOMER"。扫描计数 9, 逻辑读取次数 307, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑读取  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑读取

## 【方案2】

除了在约束列分区外，还可以在主码列分区，这样就无需在表中删除主码约束。主码都是int类型，因此可以直接用hash函数进行分区，建立分区函数及方案如下。此后重新建立新表，修改分区方案为sche\_fun\_hash并以主码对应的列为键（如ORDERS中的O\_ORDERKEY）即可。

```
drop partition function fun_hash;
CREATE PARTITION FUNCTION fun_hash (int) AS
RANGE LEFT FOR VALUES (-1073741824, 0, 1073741824)

drop partition scheme sche_fun_hash
CREATE PARTITION SCHEME sche_fun_hash AS PARTITION fun_hash
TO (GROUP1, GROUP2, GROUP3, GROUP4, GROUP5 );
```

此时运行Q3，结果如下。可见，优化效果不明显；对比IO访存次数，同样发现与baseline基本一样。究其原因，应该是由于查询中对主码的约束在表连接，此时虽然表进行了分区，但各个表的各个分区均需要进行比较、连接，因此难以发挥表分区可以迅速缩减扫描范围的优势，导致性能不佳。

```
SQL Server 执行时间:
CPU 时间 = 2044 毫秒, 占用时间 = 9629 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(10 行受影响)

表"ORDERS\_PAR"。扫描计数 8, 逻辑读取次数 22247, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 22029, 页面服务器预读读取次数 0  
表"CUSTOMER\_PAR"。扫描计数 0, 逻辑读取次数 1482, 物理读取次数 11, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, I  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 80, 页面服务器预读读取次数 0, LOB 逻辑  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑  
表"LINEITEM\_PAR"。扫描计数 8, 逻辑读取次数 104659, 物理读取次数 2, 页面服务器读取次数 0, 预读读取次数 101368, 页面服务器预读读取次数

可见，表分区并非万能的，需要根据查询在某些可以迅速缩减扫描范围的列上将表分区，方可达到效果，方案一就是一个很好的展示。

## 4、Query 4

### 4.1、基准结果

在没有任何索引、分区等的情况下运行查询，结果如下。经验证，答案正确无误。

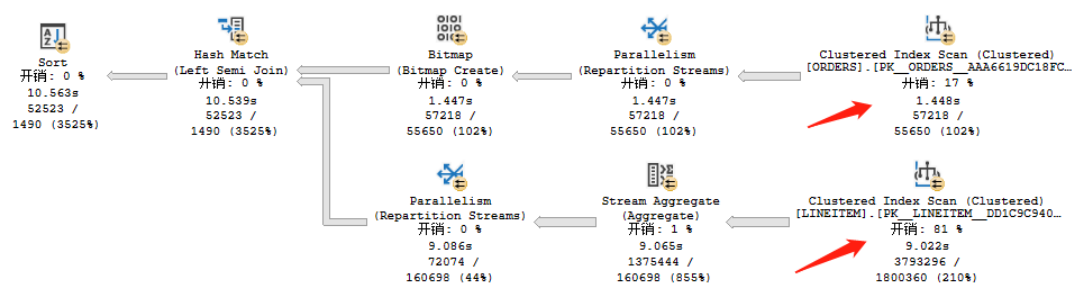
SQL Server 执行时间：  
CPU 时间 = 2031 毫秒，占用时间 = 10530 毫秒。  
SQL Server 分析和编译时间：  
CPU 时间 = 0 毫秒，占用时间 = 0 毫秒。

(5 行受影响)

表“ORDERS”。扫描计数 9，逻辑读取次数 22016，物理读取次数 0，页面服务器读取次数 0，预读读取次数 21995，页面服务器预读读取次数 0  
表“LINEITEM”。扫描计数 9，逻辑读取次数 104714，物理读取次数 2，页面服务器读取次数 0，预读读取次数 104013，页面服务器预读读取次数 0  
表“Worktable”。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB  
表“Workfile”。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB

### 4.2、索引优化分析

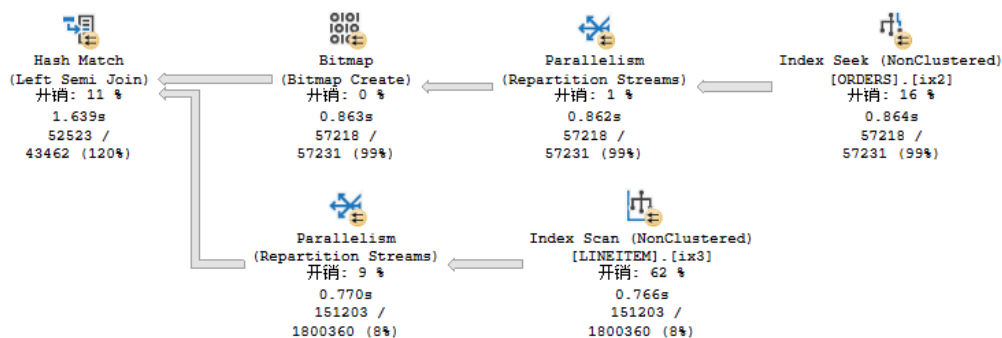
查看基准结果的执行计划，不难发现主要的时间开销来源于对ORDERS和LINEITEM两个表的聚簇索引扫描，其中LINEITEM是大头（占了80%以上），这是因为LINEITEM表本身体量最大。因此，我们的优化重点在于通过建立非聚集索引，省掉这两个聚簇扫描。



与之前方法相同，在查询的where字句中涉及的列上添加非聚集索引，同时将其其他涉及的列加入覆盖索引中。具体在本查询中，应该分别以O\_ORDERDATE、L\_ORDERKEY为键建立索引，如下。

```
create nonclustered index ix2 on ORDERS (O_ORDERDATE)
include (O_ORDERPRIORITY);
create nonclustered index ix3 on LINEITEM (L_ORDERKEY)
include (L_COMMITDATE, L_RECEIPTDATE);
```

此时再执行原查询，执行计划如下。可见，两次表扫描都变成了非聚集索引扫描，时间开销也因此显著减少。



再看执行性能统计，可见总执行时间上确实降低了不少，仅为之前的约1/6；IO方面也几乎同比例降低，这说明索引的建立导致需要操作的表变小、访存次数降低，因此性能提升。

```
SQL Server 执行时间:
CPU 时间 = 592 毫秒, 占用时间 = 1750 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(5 行受影响)

表"ORDERS"。扫描计数 9, 逻辑读取次数 4085, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 4070, 页面服务器预读读取次数 0, 表"LINEITEM"。扫描计数 9, 逻辑读取次数 14893, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 10466, 页面服务器预读读取次数 0, 表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 表"Workfile"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB

### 4.3、表分区优化分析

#### 【方案1】

注意到查询中对ORDERS表的ORDERDATE有date数值上的约束, 因此考虑根据ORDERDATE进行分区, 这样即可在查询时迅速缩小搜索范围, 起到加速的效果。对于LINEITEM表, 由于它依靠ORDERKEY位域与ORDERS表连接, 因此可以考虑在其ORDERKEY位域上添加hash分区。

按照上述分析, 新建LINEITEM和ORDERS表, 且分别在sche\_fun\_hash( L\_ORDERKEY )和 tpch\_partition\_scheme\_q3date( O\_ORDERDATE )上分区。注意, 这两个分区方案和Q3中的相同, 因此不再赘述详细的代码。详细代码也可见提交代码中q4.sql文件。

此时重新执行Q4, 在新的LINEITEM和ORDERS表上操作, 结果如下。可见, 相比baseline, 分区后查询的性能略有提升, IO次数也略有减少, 二者变化的比例基本同步, 这说明表分区带来的提升基本上源自于: 分区使得查询搜索范围迅速缩小, 减少IO进而提高性能。

```
SQL Server 执行时间:
CPU 时间 = 1843 毫秒, 占用时间 = 7653 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(5 行受影响)

表"ORDERS\_PAR"。扫描计数 12, 逻辑读取次数 17522, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 17522, 页面服务器预读读取次数 0, 表"LINEITEM\_PAR"。扫描计数 9, 逻辑读取次数 104634, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 95616, 页面服务器预读读取次数 0, 表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑读表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑读表"Workfile"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑读

这一结果固然说明表分区有效, 但一方面效果远不如索引, 另一方面也没有发挥表分区并行化的优势, 因此可以尝试其他分区方案。

#### 【方案2】

由于LINEITEM和ORDERS要在ORDERKEY上连接, 因此考虑将两个表均在这个位域上分区, 这样连接时可以通过并行化提高性能。具体地, 在新建两个表时均通过 on sche\_fun\_hash( ORDERKEY ) 进行hash分区。此时再执行查询, 结果如下。

```
SQL Server 执行时间:
CPU 时间 = 1029 毫秒, 占用时间 = 7468 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(5 行受影响)

表"LINEITEM\_PAR"。扫描计数 9, 逻辑读取次数 103541, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 97795, 页面服务器预读读取次数 0, 表"ORDERS\_PAR"。扫描计数 9, 逻辑读取次数 21934, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 21843, 页面服务器预读读取次数 0, LOB 表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑读表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOB 逻辑读

可见, IO次数略有增多, 占用时间略有减少。这较为符合预期, 说明此时的分区主要依靠并行化提高性能。但是可以注意到, 此处的变化较小, 实际上难以排除是偶然因素、系统自身因素导致的结果, 所以我们无法100%下定结论。但无论如何, 可以发现对于Q4表分区带来的优化明显不如索引, 这说明在不同问题上表分区、索引效果不同, 需要通过实践得到更好的选择。



## 5、Query 5

### 5.1、基准结果

删除所有现有非默认索引，运行Q5，基准结果如下。

```
SQL Server 执行时间:
CPU 时间 = 1187 毫秒, 占用时间 = 25779 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(5 行受影响)

表"ORDERS"。扫描计数 9, 逻辑读取次数 22211, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 22008, 页面服务器预读读取次数 0, LOb 逻辑读取次数 0  
表"CUSTOMER"。扫描计数 9, 逻辑读取次数 3449, 物理读取次数 3, 页面服务器读取次数 0, 预读读取次数 3259, 页面服务器预读读取次数 0, LOb 逻辑读取次数 0  
表"NATION"。扫描计数 9, 逻辑读取次数 4, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOb 逻辑读取次数 0, LOb 逻辑  
表"SUPPLIER"。扫描计数 9, 逻辑读取次数 601, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 193, 页面服务器预读读取次数 0, LOb 逻辑读取次数 0, L  
表"REGION"。扫描计数 9, 逻辑读取次数 4, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOb 逻辑读取次数 0, LOb 逻辑  
表"LINEITEM"。扫描计数 46008, 逻辑读取次数 453622, 物理读取次数 11, 页面服务器读取次数 0, 预读读取次数 101496, 页面服务器预读读取次数 0, LOb 逻辑  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOb 逻辑读取次数 0, LOb  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, LOb 逻辑读取次数 0, LOb

经检验，所得查询结果正确无误。

### 5.2、索引优化分析

Q5查询涉及一个长链的表连接，连接路径为：ORDER\_VALUE – ORDERS – (LINEITEM & CUSTOMER) – SUPPLIER – NATION – REGION – REGION\_VALUE，其中XX\_VALUE代表一个具体约束值。可以发现，若要构建索引，实际上有两种可能的方式——从左到右，以左侧的列为索引键，覆盖右侧的列在索引值中；或者反之，从右到左，以右侧为键，以左侧为值，如下分别尝试两种方案。

#### 【方案1】

首先尝试从右到左建立索引，例如对于CUSTOMER表，它右侧通过NATIONKEY与SUPPLIER连接，左侧通过CUSTKEY与ORDERS连接，因此可以以NATIONKEY为键以CUSTKEY为值构建索引，由于CUSTKEY为主码，这里无需显式的写入覆盖索引中。其他表的索引同理，如下所示。

```
create nonclustered index ix4 on ORDERS(O_ORDERKEY) include (O_ORDERDATE, O_CUSTKEY);
create nonclustered index ix3 on CUSTOMER(C_NATIONKEY);

create nonclustered index ix5 on LINEITEM(L_SUPPKEY)
include (L_ORDERKEY, L_EXTENDEDPRICE, L_DISCOUNT);

create nonclustered index ix3 on SUPPLIER(S_NATIONKEY);
create nonclustered index ix3 on NATION(N_REGIONKEY);
create nonclustered index ix2 on REGION(R_NAME);
```

此时再来执行Q5，结果如下。可见，IO次数及总体占用时间均显著下降，这证明添加索引可以显著提高此查询的性能。

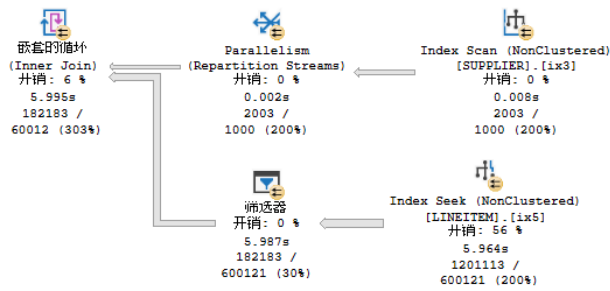
```
SQL Server 执行时间:
CPU 时间 = 1077 毫秒, 占用时间 = 6591 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

(5 行受影响)

表"SUPPLIER"。扫描计数 9, 逻辑读取次数 55, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0  
表"ORDERS"。扫描计数 9, 逻辑读取次数 3163, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 3191, 页面服务器预读读取次数 0  
表"CUSTOMER"。扫描计数 9, 逻辑读取次数 267, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 328, 页面服务器预读读取次数 0  
表"NATION"。扫描计数 9, 逻辑读取次数 4, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, L  
表"REGION"。扫描计数 4, 逻辑读取次数 4, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, L  
表"LINEITEM"。扫描计数 2003, 逻辑读取次数 21958, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 11125, 页面服务器预  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0

进一步查看执行计划，发现由于LINEITEM表最大，对其的扫描搜索是开销最大的操作，而建立LINEITEM

的索引后，其上的搜索从聚簇索引扫描变成了非聚集索引扫描，性能因而显著提升



## 【方案2】

此外，正如上面讨论，还可以从左到右建立索引，与之前的思路相似，建立的索引如下。

```
create nonclustered index ix3 on ORDERS(O_ORDERDATE) include (O_ORDERKEY, O_CUSTKEY);
create nonclustered index ix2 on CUSTOMER(C_CUSTKEY) include (C_NATIONKEY);

create nonclustered index ix4 on LINEITEM(L_ORDERKEY)
include (L_SUPPKEY, L_EXTENDEDPRICE, L_DISCOUNT);

create nonclustered index ix2 on SUPPLIER(S_SUPPKEY) include (S_NATIONKEY);
create nonclustered index ix2 on NATION(N_NATIONKEY) include (N_REGIONKEY);
create nonclustered index ix1 on REGION(R_REGIONKEY) include (R_NAME);
```

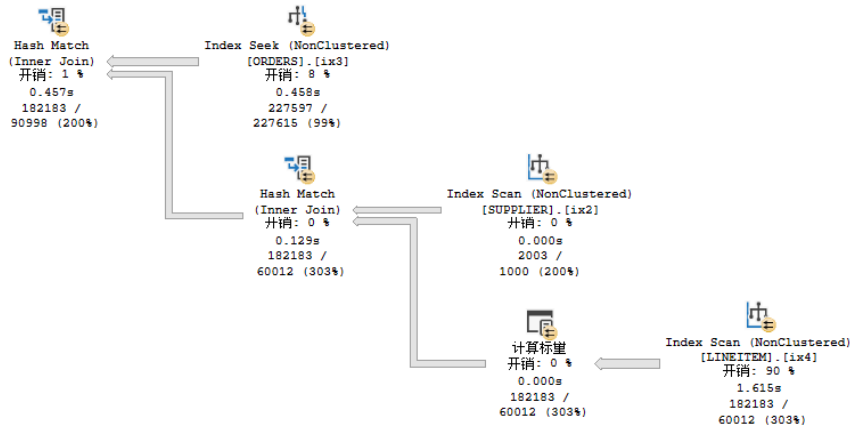
此时执行查询的性能如下所示。可见，此次的优化程度幅度更大，查询时间比Baseline快了约10倍。

SQL Server 执行时间:  
CPU 时间 = 625 毫秒, 占用时间 = 2256 毫秒。  
SQL Server 分析和编译时间:  
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。

(5 行受影响)  
表"LINEITEM"。扫描计数 9, 逻辑读取次数 26860, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 22281, 页面服务器预读读取次数 0  
表"SUPPLIER"。扫描计数 9, 逻辑读取次数 55, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 逻辑  
表"ORDERS"。扫描计数 9, 逻辑读取次数 2232, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 2250, 页面服务器预读读取次数 0, Lob 逻辑  
表"CUSTOMER"。扫描计数 9, 逻辑读取次数 261, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 272, 页面服务器预读读取次数 0, Lob 逻辑  
表"NATION"。扫描计数 9, 逻辑读取次数 4, 物理读取次数 1, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 逻辑  
表"REGION"。扫描计数 3, 逻辑读取次数 4, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 逻辑  
表"Worktable"。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, 页面服务器预读读取次数 0, Lob 逻辑

同时可以发现，IO次数与方案一差别不大，CPU时间显著缩短，总占用时间也显著缩短。这可能是因为在Q5中，左侧的VALUE约束比右侧VALUE约束强，因此在左侧建立索引时，对ORDERS表的约束即可显著缩小之后表连接的搜索范围，因而让CPU执行时间缩短。

再看执行计划，可以发现对于LINEITEM（右下角）的索引扫描时间进一步缩短。这说明在考虑索引优化时，应该优先想怎么优化最大的表的扫描，这样对于总体结果的提升是最显著的。



### 5.3、表分区优化分析

借鉴在索引优化中的结果，按照表的连接顺序从ORDERS向REGION方向建立索引效率更高；因此，在表分区时，也可以优先依据这些key来对表分区。具体地，分区依据如下表所示，其中涉及到的两种分区方案中，tpch\_partition\_scheme\_q3date用于date类型数据的均匀分区，sche\_fun\_hash为哈希分区，应用于int型数据的均匀分区；二者在之前的Query (Q3) 已经详述，不再赘述。

按照这一方式将所有表进行分区，详细代码见q5.sql。

表名	分区键	分区方案
ORDERS	O_ORDERDATE	tpch_partition_scheme_q3date
LINEITEM	L_ORDERKEY	sche_fun_hash
CUSTOMER	C_CUSTKEY	sche_fun_hash
SUPPLIER	S_SUPPKEY	sche_fun_hash
NATION	N_NATIONKEY	sche_fun_hash
REGION	R_REGIONKEY	sche_fun_hash

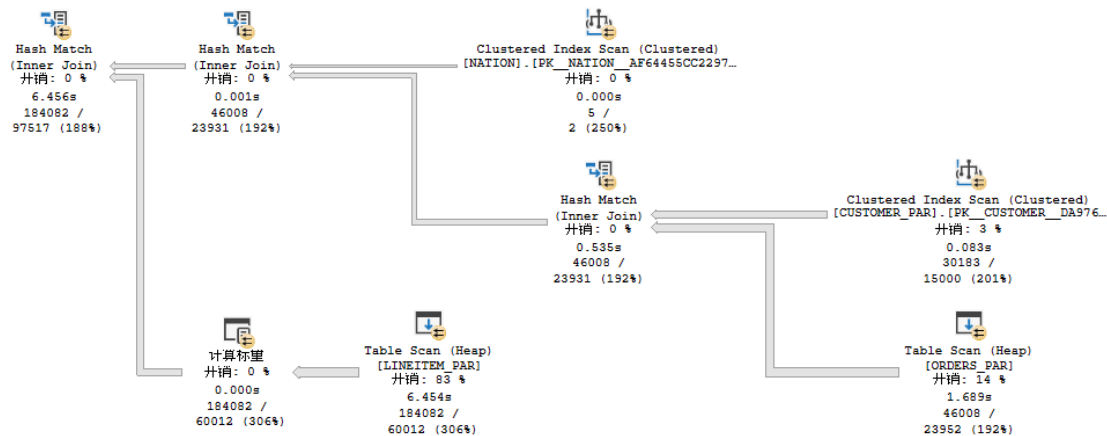
在建立分区的新表上进行查询，性能如下。可见，IO次数和执行时间均较baseline显著减少，总体占用时间从25s减少到了8s，说明表分区的优化效果明显，但总体仍不如索引的效果。

SQL Server 执行时间：  
CPU 时间 = 892 毫秒，占用时间 = 8363 毫秒。  
SQL Server 分析和编译时间：  
CPU 时间 = 0 毫秒，占用时间 = 0 毫秒。

(5 行受影响)

表“Worktable”。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑读取次数 0  
表“LINEITEM\_PAR”。扫描计数 9，逻辑读取次数 103541，物理读取次数 0，页面服务器读取次数 0，预读读取次数 103457，页面服务器预读读取次数 0，LOB 逻辑读取次数 0  
表“ORDERS\_PAR”。扫描计数 12，逻辑读取次数 17524，物理读取次数 0，页面服务器读取次数 0，预读读取次数 17524，页面服务器预读读取次数 0，LOB 逻辑读取次数 0  
表“CUSTOMER\_PAR”。扫描计数 9，逻辑读取次数 3449，物理读取次数 2，页面服务器读取次数 0，预读读取次数 1152，页面服务器预读读取次数 0，LOB 逻辑读取次数 0  
表“NATION\_PAR”。扫描计数 8，逻辑读取次数 8，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑读取次数 0  
表“SUPPLIER\_PAR”。扫描计数 8，逻辑读取次数 601，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑读取次数 0  
表“REGION\_PAR”。扫描计数 9，逻辑读取次数 8，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑读取次数 0  
表“Worktable”。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页面服务器预读读取次数 0，LOB 逻辑读取次数 0

为进一步分析表分区性能不如索引的原因，查看执行计划中占用时间最多的部分，如下所示。可见，对于LINEITEM表的全表扫描占用时间最长，这是因为LINEITEM表最大，而在其上没有建立专门针对于此查询的非聚集索引（如上一节中的ix4）；因此，需要扫描整个庞大的LINEITEM表，这一开销本身就至少6.5s，因此最终查询时间显著高于索引的2s。



对于本查询，对LINEITEM的扫描是必要的，表分区本质上只能对表检索优化，对于全表扫描并不友好（除非各个分区分布在不同的物理盘上，且IO性能相近，但本实验没有这一条件），因此添加索引的方式是更加合适的。当然，作为代价，索引引入了额外的空间开销，这一点不如表分区。因此，在优化查询时，我们需要考虑两点：①查询是否本质上就需要全表扫描对于某种优化方式更友好的操作，此时就应该优先使用那种优化，②与时间开销相对应的，是空间开销，索引的空间开销大于表分区，这在空间紧张时也需要考虑；对于此查询，表分区虽然优化没有那么明显，但相比基准已有很大提升，在空间紧张时也许是比索引更好的优化方案。

#### 5.4、物化视图优化分析

##### 【方案1】

沿用Q1的思路，可以尽可能地将可以被物化视图所计算、存储的数据放入物化视图中，这样实际上原query语句只需要读取物化视图并执行sort（实现order by）即可。

具体地，按下面代码框中的方式先建立视图dbo.query5，再建立dbo.query5上的唯一性聚簇索引ix5，这样query5就被物理地存储于磁盘上了。值得注意的是，这里视图select列除了Q5涉及的2列外，还包括count\_big(\*)列，这是因为SQL\_SERVER规定所有带group by的视图，必须有count\_big(\*)的select列，否则会报如下的错误；增加这一列并不会带来太多额外开销，但是可以成功建立物化视图。

消息 10138, 级别 16, 状态 1, 第 196 行  
无法对视图 'TPCH.dbo.query5' 创建 索引, 因为其选择列表对 COUNT BIG 的使用方法不正确。请考虑在选择列表中添加 COUNT BIG (\*)。

```
create view dbo.query5
with schemabinding
as
    SELECT N_NAME, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE, COUNT_BIG(*) TOTAL_CNT
FROM dbo.CUSTOMER, dbo.ORDERS, dbo.LINEITEM, dbo.SUPPLIER, dbo.NATION, dbo.REGION
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY AND L_SUPPKEY = S_SUPPKEY
AND C_NATIONKEY = S_NATIONKEY AND S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY
AND R_NAME = 'ASIA' AND O_ORDERDATE >= convert(DATETIME, '1994-01-01', 102)
AND O_ORDERDATE < DATEADD(YY, 1, convert(DATETIME, '1994-01-01', 102))
GROUP BY N_NAME;

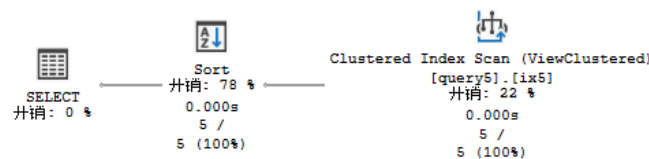
create unique clustered index ix5
on dbo.query5( N_NAME );
```

此时再执行Q5，性能表现如下。可以发现，执行时间已经缩短到几乎没有的量级了，IO数据及执行计划也印证了这一点——由于物化视图将所有的结果都已经计算出，最后只需要做一个排序，而结果长度很短（只有5），所以总执行时间、IO开销几乎可以忽略不计。与之相对的，query的分析和编译时间相较执行时间竟然更长，这或许是因为为了分析出“Q5的执行可以利用dbo.query5”，需要更多的分析和运算。但不论如何，此方式构建的物化视图带来了很大的性能提升。

SQL Server 分析和编译时间：  
CPU 时间 = 140 毫秒，占用时间 = 224 毫秒。

SQL Server 执行时间：  
CPU 时间 = 0 毫秒，占用时间 = 74 毫秒。

(5 行受影响)  
表“Worktable”。扫描计数 0，逻辑读取次数 0，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0。  
表“query5”。扫描计数 1，逻辑读取次数 2，物理读取次数 0，页面服务器读取次数 0，预读读取次数 0，页



## 【方案2】

上述结果带来的提升固然很大，但这一物化视图实际上并不具有普适性——它与Q5太过接近，这样虽然Q5可以使用，它对其他查询带来优化的可能性实际上很小。因此，可以观察Q5，得到一个更“基础”的物化视图，应用于Q5，分析它的效果，往往更有实际意义。

观察Q5不难发现，它的基本是六个表的连接，在连接的两头做数值上的约束，这样得到的表再运行 group by、select中的运算、order by操作。因此，“六表连接”实际上是一个基础操作，如果我们将物化视图建立为表连接的结果，这一视图将会是普适的，任何涉及表连接，且连接的表为这六个表的子集的查询均可使用此视图进行优化。所以利用如下的代码建立视图，

```
create view dbo.query5_2
with schemabinding
as
    SELECT N_NAME, L_EXTENDEDPRICE, L_DISCOUNT, O_ORDERDATE, R_NAME
    FROM dbo.CUSTOMER, dbo.ORDERS, dbo.LINEITEM, dbo.SUPPLIER, dbo.NATION, dbo.REGION
    WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY AND L_SUPPKEY = S_SUPPKEY
    AND C_NATIONKEY = S_NATIONKEY AND S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY;
```

再在dbo.query5\_2上建立聚簇索引ix1，这样物化视图就建立完毕了。

```
create unique clustered index ix1
on dbo.query5_2( N_NAME, O_ORDERDATE, L_EXTENDEDPRICE );
```

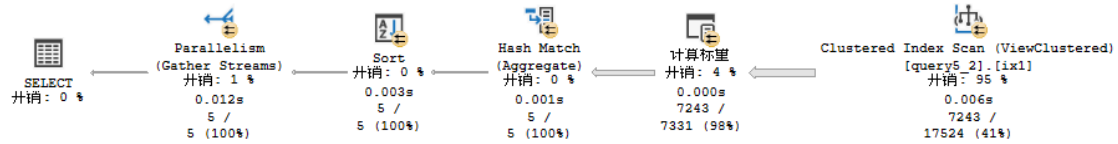
此后再执行Q5原查询，性能表现如下。可见，虽然此物化视图提前完成的任务并不多，但其性能表现却出乎意料的好——其总体占用时间及分析编译时间与方案一中几乎相同，虽然由于此时并没有“一步到位”，IO次数多了一些，但这一数字仍然较小，对总时间的影响可以忽略不计。观察执行计划，可以证实Q5查询确实利用了我们建立的query5\_2的物化视图。



SQL Server 分析和编译时间:  
CPU 时间 = 156 毫秒, 占用时间 = 216 毫秒。

SQL Server 执行时间:  
CPU 时间 = 16 毫秒, 占用时间 = 81 毫秒。

(5 行受影响)  
表“query5\_2”。扫描计数 9, 逻辑读取次数 2391, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0  
表“Worktable”。扫描计数 0, 逻辑读取次数 0, 物理读取次数 0, 页面服务器读取次数 0, 预读读取次数 0, ...



通过这一实验，我们可以发现，建立物化视图未必要让视图与查询尽可能“接近”、“一步到位”。相反，我们应该针对查询开销较大的基础步骤（例如表连接、表扫描等）的结果建立物化视图，这样一方面，物化视图所没有完成的任务（如排序、分组等），即便交给每次查询现场完成，也不会有多大的开销，性能仍能保持在较高水平，另一方面，这样建立的物化视图可以应用于更多查询中，以固定的额外空间开销让更多查询性能提高，这是利大于弊的。

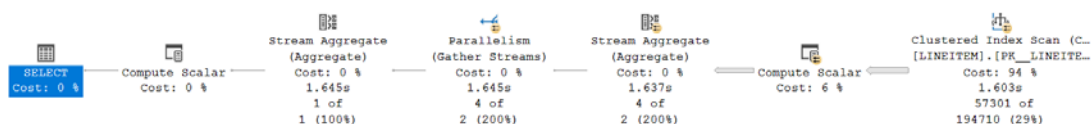
总之，通过物化视图，确实是可以让查询的性能有显著提升，幅度高于索引、表分区。

## 6、Query 6

### 6.1、基准结果

首先，在没有任何索引的基础上运行查询，检查运行时间、执行计划及访存信息，如下。经验证，查询答案正确。

(1 row affected)  
表 'LINEITEM'。扫描计数 5, 逻辑读取 52000 次, 物理读取 0 次, 预读 51965 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
(1 row affected)  
SQL Server 执行时间:  
CPU 时间 = 2577 毫秒, 占用时间 = 1882 毫秒。  
SQL Server 执行时间:  
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。



### 6.2、索引优化分析

从执行计划上看，对于LINEITEM表的聚簇索引扫描占据了大量的时间，事实上query 6对应的查询仅针对单表进行，且带有聚集。具体分析一下查询的内容：对于一年内，订单折扣在0.05~0.07之间，且订单数量小于24的这部分订单计算其总价与折扣率乘积的总和。更直观来说，如果对这一年的所有总量小于24的订单不打折，那么可以为公司带来多少盈利。从上面的分析可以看到，查询只在单表LINEITEM上进行，查询的条件依赖多个字段：[L\_DISCOUNT],[L\_QUANTITY],[L\_SHIPMENTDATE]；而对于LINEITEM表的扫描目前是基于主码的聚簇索引进行的，这也是为什么对表的扫描占据了大量时间的原因。我们可以先尝试在[L\_DISCOUNT],[L\_QUANTITY],[L\_SHIPMENTDATE]字段上建立非聚簇索引；由于查询结果只涉及到[L\_DISCOUNT],[L\_EXTENDEDPRICE]字段，非聚簇索引只需要覆盖这些字段。

```
CREATE NONCLUSTERED INDEX [Q6_IDX] ON [dbo].[LINEITEM]
([L_SHIPDATE],[L_DISCOUNT],[L_QUANTITY]) INCLUDE
([L_EXTENDEDPRICE])
```

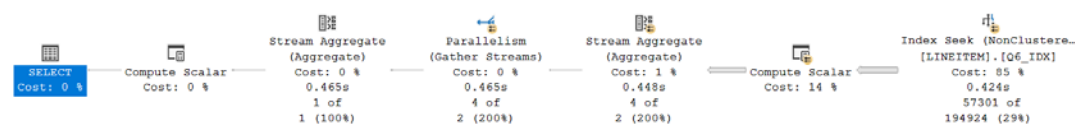
建立索引后再次执行Q6，结果如下。

```
(1 row affected)
表 'LINEITEM'. 扫描计数 5, 逻辑读取 7679 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。

(1 row affected)

SQL Server 执行时间:
  CPU 时间 = 843 毫秒, 占用时间 = 711 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 = 0 毫秒。
```



可见，时间开销进一步从1.9s减少到0.7s，原因主要是执行计划中从全表扫描变为了利用上面建立的非聚簇索引的扫描，从而大大减少了对LINEITEM表的读取次数，进而减少了扫描时间。

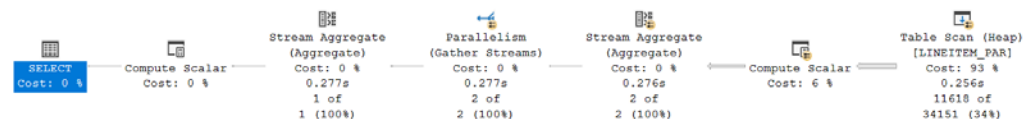
### 6.3、表分区优化分析

接下来考虑表分区带来的优化效果。Q6对应的查询约束条件很多，可以把日期、折扣和数量几个字段作为分区字段，从而进一步减少访存的开销。对日期进行分区是最自然的想法，因为大多数查询都会以日期作为约束条件，考虑一定时间范围内的数据。

```
(1 row affected)
表 'LINEITEM'. 扫描计数 5, 逻辑读取 7679 次, 物理读取 0 次, 预读 4 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。

(1 row affected)

SQL Server 执行时间:
  CPU 时间 = 875 毫秒, 占用时间 = 592 毫秒。
```



相比baseline减少了将近2/3的时间。值得注意的是扫描表花费的开销显著减小，因为我们这次的查询涉及的日期被限定在一年的范围内；而我们的分区依据是把临近的几年放在一个分区中。这样一来读取一年的数据其实仅需要访问一个分区。对比Q1查询，以相同的依据建立分区，对查询的优化效果却截然不同，其原因是Q6对分区字段加入了更强的约束；而Q1则相对松散，即使进行分区，也需要从多个分区中读取。

以日期进行分区是十分常见的方法，因为我们一般会比较关心某个时间区间的数据；因此日期会经常地出现在查询的限制条件中，而如果数据库依据日期进行分区就可以减少这些查询需要访问的存储规模，提高查询性能。

### 6.4、物化视图

结合前面的例子，创建物化视图往往可以非常大幅度地提高查询效率；我们也进行一些物化视图的尝试

```

create view dbo.query6
with schemabinding
as
    SELECT
        SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS REVENUE,
        count_big(*) as count_order
    FROM dbo.LINEITEM
    WHERE L_SHIPDATE >= CONVERT(DATETIME, '1994-01-01', 102)
    AND L_SHIPDATE < dateadd(yy, 1, CONVERT(DATETIME, '1994-01-01', 102))
    AND L_DISCOUNT BETWEEN .06 - 0.01 AND .06 + 0.01 AND L_QUANTITY < 24;

```

#### 6.4.1、直接创建保存结果的视图

由于在创建视图时就已经对所有约束条件加以了限定，而视图中其实也仅有一条数据，在此基础上执行的查询几乎不用花费时间，效率大大提高：

```

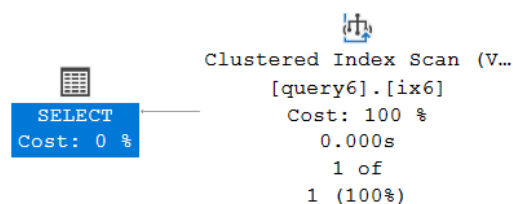
(1 row affected)
表 'query6'. 扫描计数 1, 逻辑读取 2 次, 物理读取 0 次, 预读 0 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。

(1 row affected)

SQL Server 执行时间:
    CPU 时间 = 15 毫秒, 占用时间 = 34 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 = 0 毫秒。

```



但我们也意识到，这种类型的优化意义非常小，只有在所有约束条件都完全一致的情况下才能起到提高效率的作用。于是，希望采用另一种物化视图的方式，使得它至少可以应对一种类型的查询。

#### 6.4.2、更普适的物化视图方式

```

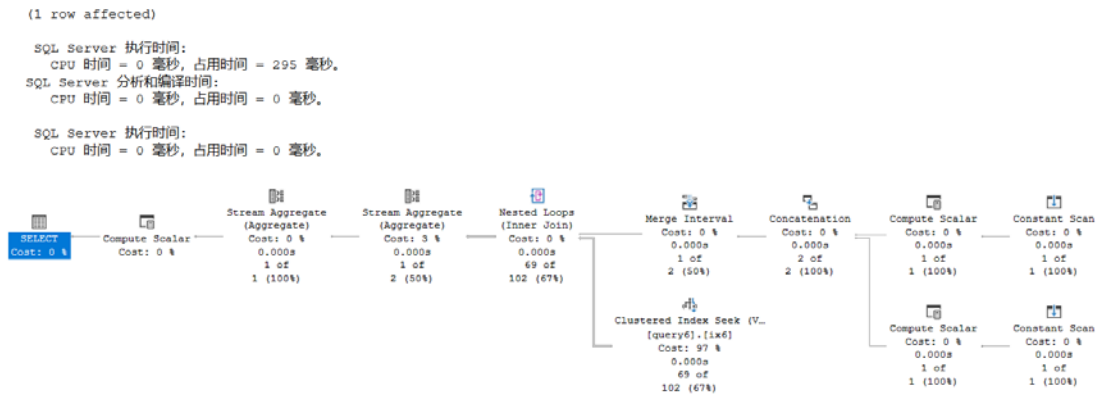
create view dbo.query6
with schemabinding
as
    SELECT
        SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS REVENUE,
        L_DISCOUNT,
        L_QUANTITY,
        count_big(*) as count_order
    FROM dbo.LINEITEM
    WHERE L_SHIPDATE >= CONVERT(DATETIME, '1994-01-01', 102)
    AND L_SHIPDATE < dateadd(yy, 1, CONVERT(DATETIME, '1994-01-01', 102))

```

这里仍然限定了日期范围，主要是考虑到对于这种假象盈利的查询往往是以年的维度进行统计的。这次需要在L\_DISCOUNT 和 L\_QUANTITY上建立唯一聚集索引：

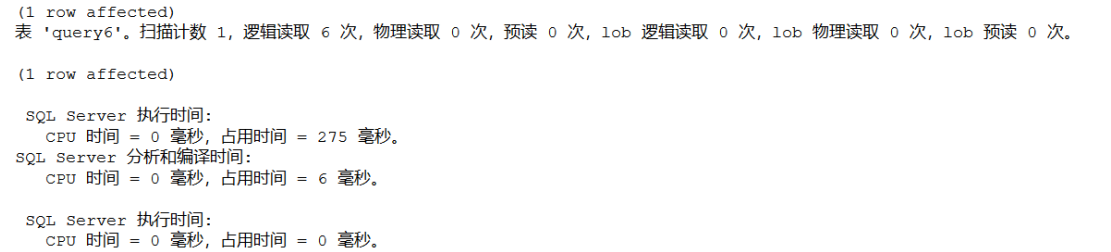
```
create unique clustered index ix6 on dbo.query6 (L_DISCOUNT,L_QUANTITY);
```

再执行原来的查询，效果虽然不及上一种物化视图方式，但比baseline的还是有几十倍的提高：



从执行计划中看到物化视图大大减少了对数据库的读取次数，提高了查询效率。我们再尝试进行一个类似的查询，这里改变对折扣率和交易量的约束条件，再执行一个查询：

```
WHERE L_DISCOUNT BETWEEN .04 - 0.01 AND .04 + 0.01 AND L_QUANTITY < 14;
```



效率保持在较高水平。从这个例子上我们意识到，物化视图/任何针对数据库性能的优化都应该兼顾效率与适用性；如果针对的查询范围太窄，总得来看收益并不会很高。

## 7、Query 7

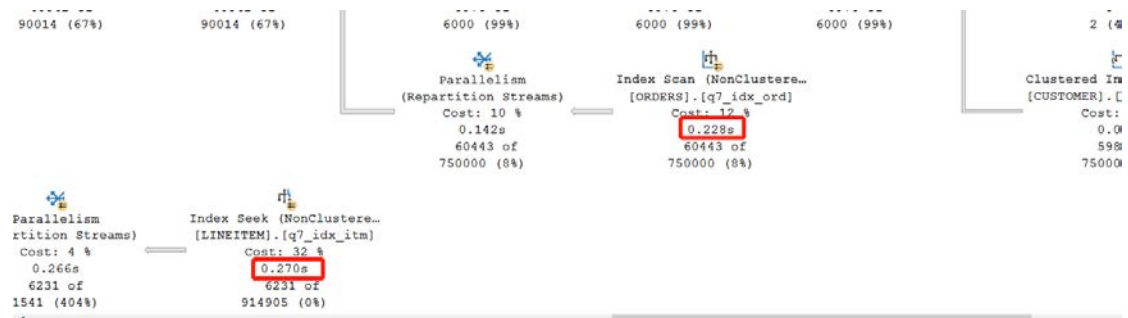
### 7.1、基准结果

首先，在没有任何索引的基础上运行查询，检查运行时间、执行计划及访存信息，如下。经验证，查询答案正确。





优化效果非常显著, 仅仅是添加索引就把逻辑读取次数减小为添加前的 1/10 左右; 运行时间也从 4.7s 减小为 1.0s. 从执行计划上来看, 原来占据大头的对 LINEITEM 表的读取从原先 3.5s 减小为 0.3s, 索引的效果非常显著:



再进一步分析其他占用时间较多的项目, 我们看到只要涉及到与表 LINEITEM 的连接操作都会比较耗费时间, 于是想通过非聚簇索引来减少连接的开销。分析查询语句, LINEITEM 表与其他表连接时用到两个字段: [L\_SUPPKEY], [L\_ORDERKEY]; 之前为了提高扫描速度的时候加的索引只放在了日期, 对连接的优化效果有限; 我们再看 LINEITEM 上添加一个新的索引。

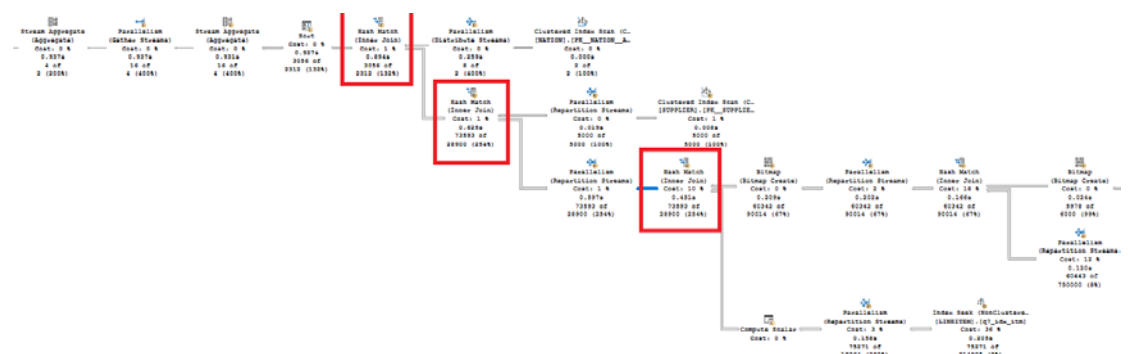
```

DROP INDEX if EXISTS q7_idx_itm_2 on LINEITEM;
CREATE NONCLUSTERED INDEX q7_idx_itm_2
ON [dbo].[LINEITEM] ([L_SUPPKEY], [L_ORDERKEY])
INCLUDE ([L_EXTENDEDPRICE], [L_DISCOUNT])

DROP INDEX if EXISTS q7_idx_cus on CUSTOMER;
CREATE NONCLUSTERED INDEX q7_idx_cus
ON [dbo].[CUSTOMER] ([C_NATIONKEY])

```

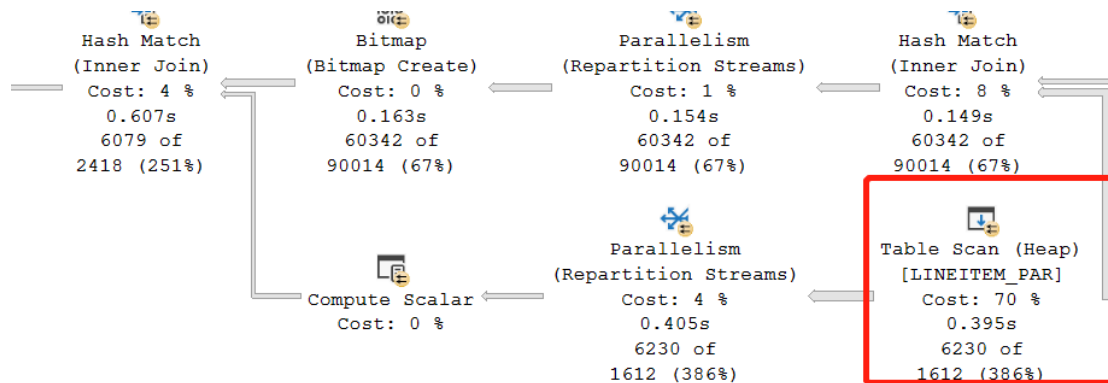
然而实际的效果非但没有提高, 反而有所下降: 执行时间从 1.0s 又上升到了 1.4s 这样的结果是由于 sql server 基于新的索引改变了查询结构:



我们发现 LINEITEM 更早地参与了与其他表的连接, 于是更大的数据规模需要伴随更多的运算; 另外, 新增加的索引项并没有明显提高连接的效率; 综合来看新的索引反而起了适得其反的作用。

### 7.3、分区优化

这里分区优化的效果比较有限, 与之前的思路相同, 对 LINEITEM 表依据日期进行分区, 从而降低查询涉及到的数据规模, 减少扫描时间:



对数据分区存储以后，扫描时间从 3s 减少到 0.4s.

## 8、Query 8

### 8.1、基准结果

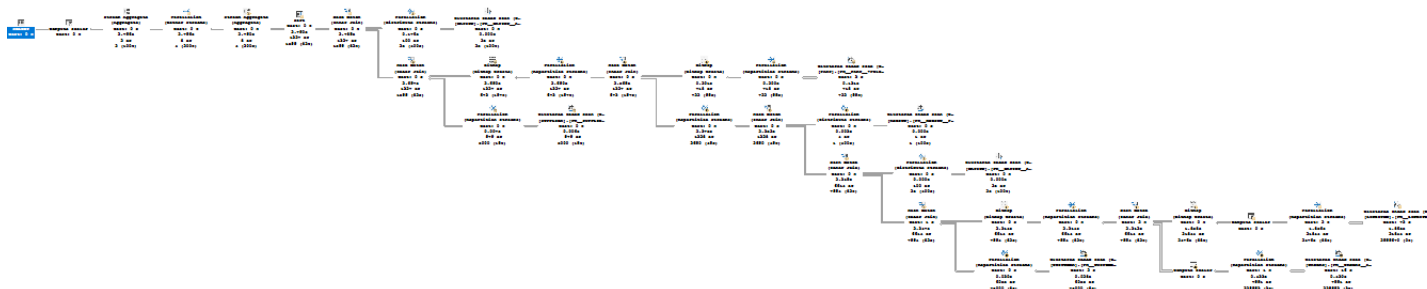
首先，在没有任何索引的基础上运行查询，检查运行时间、执行计划及访存信息，如下。经验证，查询答案正确。

(2 rows affected)

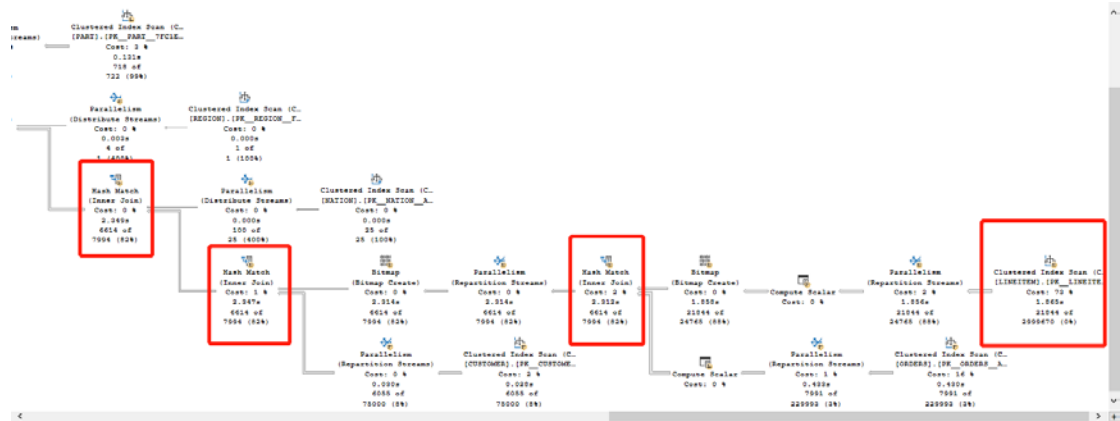
表 'NATION'. 扫描计数 2, 逻辑读取 4 次, 物理读取 0 次, 预读 0 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'PART'. 扫描计数 5, 逻辑读取 1838 次, 物理读取 11 次, 预读 1248 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'REGION'. 扫描计数 1, 逻辑读取 2 次, 物理读取 0 次, 预读 0 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'LINEITEM'. 扫描计数 5, 逻辑读取 52000 次, 物理读取 753 次, 预读 51896 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'ORDERS'. 扫描计数 5, 逻辑读取 11012 次, 物理读取 32 次, 预读 11012 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'Workfile'. 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 'CUSTOMER'. 扫描计数 5, 逻辑读取 1643 次, 物理读取 0 次, 预读 1643 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'SUPPLIER'. 扫描计数 5, 逻辑读取 101 次, 物理读取 0 次, 预读 101 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 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。

(1 row affected)

SQL Server 执行时间:  
 CPU 时间 = 1371 毫秒, 占用时间 = 3004 毫秒。  
 SQL Server 分析和编译时间:  
 CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。



仔细分析执行计划，可以发现：



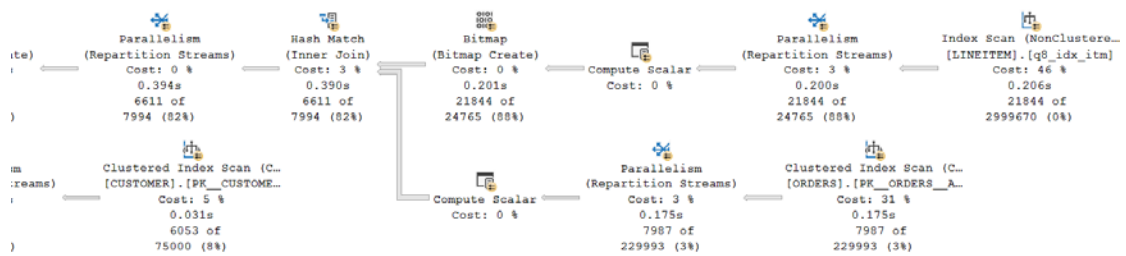
分析执行流程，对 LINEITEM 表的扫描以及后续的连接操作都占据了大量时间。可以想见，如果在 LINEITEM 上建立索引应该可以大幅提高查询效率。

按照这个思路添加索引：

```

DROP INDEX if EXISTS q8_idx_itm on LINEITEM;
CREATE NONCLUSTERED INDEX q8_idx_itm
ON [dbo].[LINEITEM]
([L_SUPPKEY],[L_ORDERKEY],[L_PARTKEY])
INCLUDE ([L_EXTENDEDPRICE],[L_DISCOUNT])

```



(2 rows affected)

表 'NATION'. 扫描计数 2, 逻辑读取 4 次, 物理读取 1 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 'PART'. 扫描计数 5, 逻辑读取 1838 次, 物理读取 0 次, 预读 1845 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'REGION'. 扫描计数 1, 逻辑读取 2 次, 物理读取 1 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 'LINEITEM'. 扫描计数 5, 逻辑读取 14900 次, 物理读取 0 次, 预读 9845 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'ORDERS'. 扫描计数 5, 逻辑读取 11012 次, 物理读取 0 次, 预读 11012 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'Workfile'. 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 '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 'CUSTOMER'. 扫描计数 5, 逻辑读取 1643 次, 物理读取 0 次, 预读 1649 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 表 'SUPPLIER'. 扫描计数 5, 逻辑读取 101 次, 物理读取 0 次, 预读 104 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。  
 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。

(1 row affected)

SQL Server 执行时间:  
 CPU 时间 = 530 毫秒, 占用时间 = 1054 毫秒。  
 SQL Server 分析和编译时间:  
 CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。

我们看到总的查询时间从 3.0s 减小为 1.1s; 其中对 lineitem 的扫描次数减小了 5 倍。观察到 where 条件限制了 ORDERS 表的时间范围, 且 ORDERS 表与其他表的连接操作也花费了大量的时间, 于是希望在 L\_ORDERS 字段上建立非聚簇索引。

```

CREATE NONCLUSTERED INDEX query8_idx_order
ON [dbo].[ORDERS] ([O_ORDERDATE])
INCLUDE ([O_CUSTKEY])

```

奇怪的是加入以上索引后查询时间又回到了 3s。检查执行计划发现 sql server 此时对 LINEITEM 表的扫描变为基于聚簇索引进行，即之前加在 LINEITEM 表上的索引并没有被利用。分析其原因，因为对表查询时的约束条件只加到了 REGION, ORDER 和 PART 上，于是在从 LINEITEM 上取数的时候，sql server 认为使用全表扫描的时间代价会小于使用索引。然而由于 LINEITEM 后面需要和其他具有约束的表连接，此时如果利用索引的话就可以大大提升效率。可见 sql server 选择最优执行方案的时候不一定每一次都能找到最优方案。

## 9、Query 9

### 9.1、基准结果

首先，在没有任何索引的基础上运行查询，检查运行时间、执行计划及访存信息，如下。经验证，查询答案正确。

```
(175 rows affected)
表 'SUPPLIER'. 扫描计数 5, 逻辑读取 101 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 'PART'. 扫描计数 5, 逻辑读取 1838 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 'LINEITEM'. 扫描计数 5, 逻辑读取 52000 次, 物理读取 0 次, 预读 51999 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
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 'PARTSUPP'. 扫描计数 0, 逻辑读取 14953 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 'ORDERS'. 扫描计数 5, 逻辑读取 11012 次, 物理读取 0 次, 预读 11010 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'NATION'. 扫描计数 0, 逻辑读取 350 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 '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。

(1 row affected)

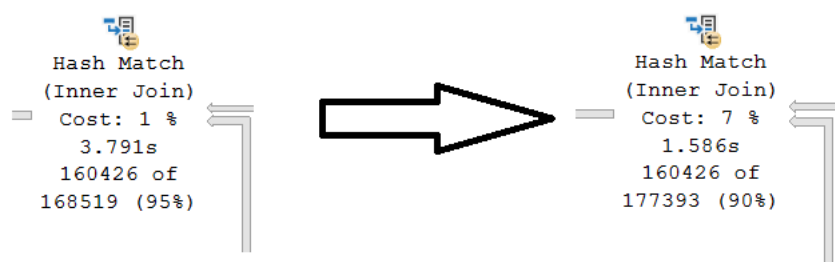
SQL Server 执行时间:
CPU 时间 = 4573 毫秒, 占用时间 = 5508 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。
```

### 9.2、索引优化

先只在 LINEITEM 表上添加索引：

```
CREATE NONCLUSTERED INDEX query9_idx_itm
ON [dbo].[LINEITEM] (L_SUPPKEY, L_PARTKEY, L_ORDERKEY)
INCLUDE (L_EXTENDEDPRICE, L_DISCOUNT, L_QUANTITY)
```

在用于和其他表关联的字段上建立索引，预期这样可以减少关联时的开销：



从执行计划上看，效率确实得到提高；总执行时间也由 5.5s 减小到 3.1s。同样在其他表上也添加索引：

```
CREATE NONCLUSTERED INDEX query9_idx_P
ON [dbo].[PART] (P_PARTKEY, P_NAME)
CREATE NONCLUSTERED INDEX query9_idx_PS
ON [dbo].[PARTSUPP] (PS_PARTKEY)
INCLUDE (PS_SUPPLYCOST)
CREATE NONCLUSTERED INDEX query9_idx_O
ON [dbo].[ORDERS] (O_ORDERKEY)
INCLUDE (O_ORDERDATE)
```

```

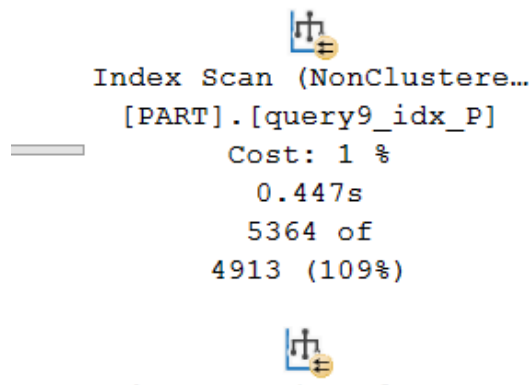
(175 rows affected)
表 'NATION'. 扫描计数 1, 逻辑读取 2 次, 物理读取 0 次, 预读 0 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'SUPPLIER'. 扫描计数 5, 逻辑读取 101 次, 物理读取 0 次, 预读 0 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'ORDERS'. 扫描计数 5, 逻辑读取 1212 次, 物理读取 0 次, 预读 389 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'PART'. 扫描计数 5, 逻辑读取 585 次, 物理读取 0 次, 预读 135 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'LINEITEM'. 扫描计数 5, 逻辑读取 18241 次, 物理读取 198 次, 预读 11565 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'Workfile'. 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 '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 'PARTSUPP'. 扫描计数 5, 逻辑读取 1381 次, 物理读取 0 次, 预读 583 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
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。

(1 row affected)

SQL Server 执行时间:
CPU 时间 = 9939 毫秒, 占用时间 = 4369 毫秒。
SQL Server 分析和编译时间:
CPU 时间 = 0 毫秒, 占用时间 = 0 毫秒。

```

我们发现效果反而变差了；利用索引的确减少了各个表的访问次数，但多表连接时的开销反而变大了。我们发现 where 子句中存在对 P\_NAME 的模糊查询；而该约束要求 P\_NAME 中包含字符串'green'并没有要求以'green'开头；这也就导致我们添加在 P\_NAME 上的索引失效，因而影响到了效率；从执行计划中我们发现：



```

Index Scan (NonClustered...)
    [PART].[query9_idx_P]
    Cost: 1 %
    0.447s
    5364 of
    4913 (109%)

```

Sql server 仍然运用了我们添加的索引，但事实上利用索引并不能缩小查询范围；相反，还丢掉了全表顺序扫描的效率，因此查询时间不降反增。认识到这个问题，我们去掉了作用在 PART 上的索引从新运行一遍：

```

(175 rows affected)
表 'NATION'. 扫描计数 1, 逻辑读取 2 次, 物理读取 1 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 'SUPPLIER'. 扫描计数 5, 逻辑读取 101 次, 物理读取 0 次, 预读 104 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'PART'. 扫描计数 5, 逻辑读取 1838 次, 物理读取 0 次, 预读 1845 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'LINEITEM'. 扫描计数 5, 逻辑读取 18241 次, 物理读取 0 次, 预读 18241 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'Workfile'. 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 '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 'ORDERS'. 扫描计数 5, 逻辑读取 1212 次, 物理读取 0 次, 预读 1226 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
表 'PARTSUPP'. 扫描计数 5, 逻辑读取 1381 次, 物理读取 0 次, 预读 1237 次, lob 逻辑读取 0 次, lob 物理读取 0 次, lob 预读 0 次。
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。

(1 row affected)

SQL Server 执行时间:
CPU 时间 = 5246 毫秒, 占用时间 = 2860 毫秒。
SQL Server 分析和编译时间:

```

时间缩短到 2.8s

补充：如果查询语句的 where 子句为 P\_NAME like '%Green' 要求以 green 结尾，依然存在无法使用索引对查询优化的问题；但可以对 P\_NAME 的反转建立索引：

```

CREATE NONCLUSTERED INDEX rev_idx
ON [dbo].[PART] (reverse(P_NAME))

```

查询时把约束条件改写为 reverse(P\_NAME) like reverse('%Green') 相当于找开头为'neerG'的项目，即可利用索引进行优化了。



### 三、总体测评

#### 准备工作

首先希望去掉所有非聚簇索引，记录没有任何优化操作时的查询效率，利用如下代码生成删除索引的sql语句：

```
EXEC sp_spaceused
SELECT
    ('drop index ' + idx.name + ' on ' +
    OBJECT_NAME(CAST(idx.object_id AS INT)) + ';') AS
    dropIndexScript
FROM sys.tables tb
    INNER JOIN sys.indexes idx ON idx.object_id =
    tb.object_id
WHERE tb.type = 'u'
    AND idx.is_unique = 0
    AND idx.name IS NOT NULL
```

#### 运行结果

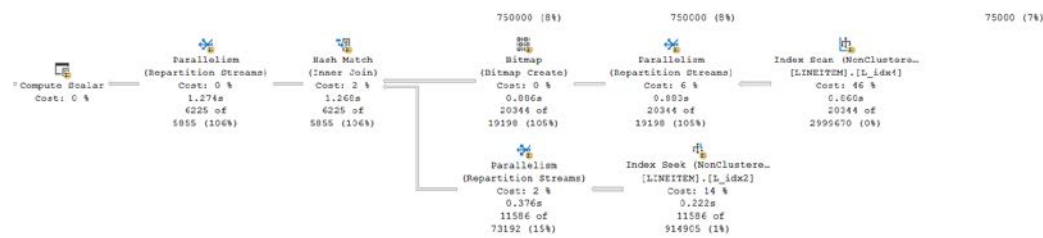
	dropIndexScript
1	drop index P_idx1 on PART;
2	drop index S_idx1 on SUPPLIER;
3	drop index PS_idx1 on PARTSUPP;
4	drop index R_idx1 on REGION;
5	drop index N_idx1 on NATION;
6	drop index N_idx2 on NATION;
7	drop index C_idx1 on CUSTOMER;
8	drop index O_idx1 on ORDERS;
9	drop index O_idx2 on ORDERS;

#### 性能比较

查询	基础结果(MS)	添加全局索引(MS)	分查询优化(MS)
Q1	8804	8710	2773
Q2	991	1374	252
Q3	2868	910	1046
Q4	3008	977	752
Q5	3639	1195	608
Q6	549	361	140
Q7	3173	20040	1199
Q8	2743	2783	604
Q9	5321	3159	1867
Q1~Q9总计	31096	39509	9240
编译时间	2261	5231	-

可以明显看到，相较于没有添加任何索引的时候，添加全局索引使得编译时间提升了一倍之多，这是意料之中的情况：如果某个表上只有聚集索引，从表中读数的时候只有扫描表一种途径；但当存在非聚集索引的时候，sql server就需要权衡使用索引来获取数据还是对全表进行扫描；而当表上存在多个非聚集索引时，还需要比较使用不同索引的效率。事实上，当某个表上存在过多索引的时候，权衡不同索引的开销甚至可能超过添加索引带来的收益。

另一方面，相比于未添加索引，添加针对全部query的全局索引在大多数case上的表现都有不同程度的提高，唯独在query7上查询时间提高了6倍。



我们发现执行计划中对LINEITEM表的扫描采用了两套不同的索引，而事实上这两套索引都不是最适合query7的；这里也反应出数据库索引优化的一个矛盾：如果希望每一条查询语句都能得到优化，就需要创建和存储大量的索引（创建和存储的过程也需要耗费大量时间、空间成本），而且真正执行查询时选择合适索引的开销也随之变大；

#### 四、总结

本实验探索了在TPC-H上进行数据库性能调优的方法，主要包括添加索引、表分区及利用物化视图。我们尝试的每种优化方式均有各自的特点，例如特别针对于query的索引往往可以显著提高性能，表分区比较普遍，不同query的优化都可以使用同一种分区，但优化幅度往往没有那么大，物化视图带来的潜在优化空间是最大的，但是由于其严格的要求，很多查询难以完全使用物化视图。

同时，不同查询也涉及到权衡的问题——索引优化幅度虽然大，但有额外的空间开销，当这些开销过大时，往往表分区会更有效；物化视图越“一步到位”，对当前query的优化通常越明显，但也失去了普适性，对其它查询往往起不到作用，此时如果物化视图的空间开销较大，往往是不值当的；另外，建立索引、物化视图还需要额外的建立时间开销，这也就代表着如果建立了这些结构但查询的频率很低，建立索引+总运行时间甚至可能慢于未添加索引时的净执行时间，因此在更加现实的场景中，添加索引带来的时间开销也是需要根据索引使用频率等情况考虑的问题。

总之，对数据库性能的优化往往是具体问题具体分析的一项任务，应熟练掌握各类优化技巧，在所给任务上多加尝试，才能得到相对合适、高效的优化策略。