**性能调优实习报告**

1700017802 耿思博 1700017815 张旻昊 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）

其次分析各查询结构特征，分析数据库参数、索引等对查询性能的影响

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TPCH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。

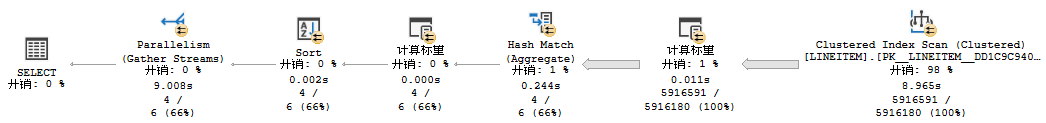
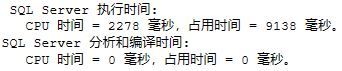
dbcc dropcleanbuffers

dbcc freeproccache

**二、各查询分析及优化**

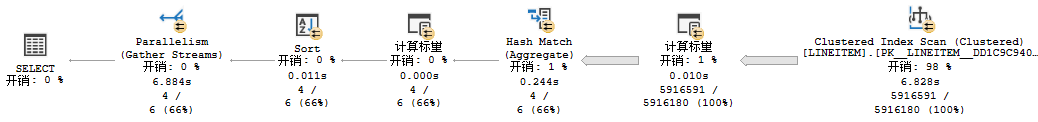
**1、Query 1**

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。

经验证，查询答案正确。

1.2、索引优化分析

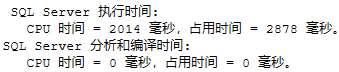
首先，由于我们的查询不存在并发的情形，可以接受read uncommitted的隔离性级别，因此先设置该隔离性级别，再次执行，所得结果如下。

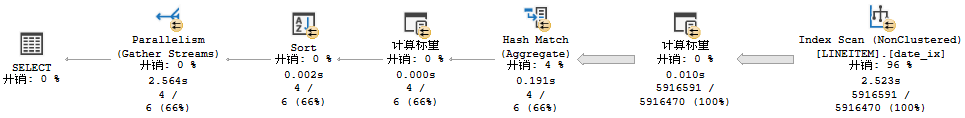
可见，时间由9s缩短至7s，但是读取次数几乎没有变化（事实上每次执行，逻辑读取次数都略有不同，存在随机因素），这符合预期——修改隔离性级别并不能减少读取次数，它仅能减小并发控制时带来的额外开销，因此总查询时间缩短。

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。





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\_q1date( DATE )

AS RANGE RIGHT

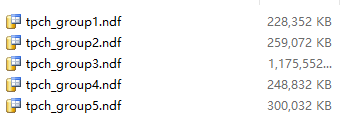
FOR VALUES( '1993-07-01','1994-10-24', '1996-03-01', '1997-06-15');

CREATE PARTITION SCHEME tpch\_partition\_scheme\_q1date

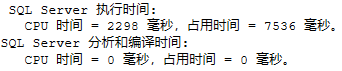
AS PARTITION tpch\_partition\_q1date

TO (GROUP1, GROUP2, GROUP3, GROUP4, GROUP5 );

最后，新创建LINEITEM\_PAR表格，定义与LINEITEM相同，只不过最后加上字句 on tpch\_partition\_scheme\_q1date( 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。



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，因此表分区的意义实际上不大——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。





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\_EXTENDEDPRICE) AS SUM\_BASE\_PRICE, SUM(L\_EXTENDEDPRICE\*(1-L\_DISCOUNT)) AS SUM\_DISC\_PRICE,

SUM(L\_EXTENDEDPRICE\*(1-L\_DISCOUNT)\*(1+L\_TAX)) AS SUM\_CHARGE,

SUM(L\_QUANTITY) AS SUM\_QTY, COUNT\_BIG(L\_QUANTITY) AS CT\_QTY,

SUM(L\_EXTENDEDPRICE) AS SUM\_PRICE, COUNT\_BIG(L\_EXTENDEDPRICE) 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 );

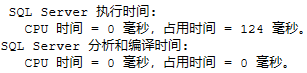
首先检验物化视图的正确性，做如下查询，相当于“包装”了query1视图，加上后处理，得到与Q1相同的语义。运行这一查询发现，结果与Q1结果（q1.out）相同，可见物化视图的语义正确。

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

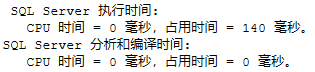
from dbo.query1 ORDER BY L\_RETURNFLAG, L\_LINESTATUS;

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。



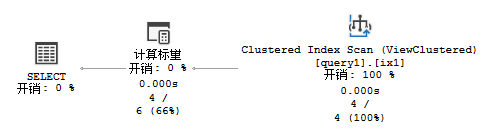


建立此物化视图后，再**查询原查询Q1**（上面只是运行了包装查询检验正确性、有效性，为了统一比较还是应该基于原查询跑一遍结果），结果如下。可见，总运行时间约为0.14s，同样显著低于baseline，达到了非常好的性能。IO结果与上面的结果相似，**这说明建立物化视图后，DBMS可以自动在查询中识别可以使用物化视图的部分，进行优化，提高性能**。





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。



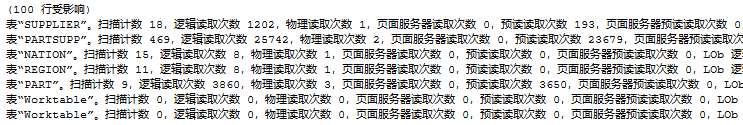
然而，物化视图也拥有一定的问题：如果它与查询极为接近（例如上面的例子），则它可以对相应的查询有极大的优化，但对于其他大多数查询则难以用到；如果要让它用到更多的查询中，则优化幅度就没有那么明显，这需要用户权衡；另外，物化视图同样需要额外的开销存储视图，与索引相同，使得其在空间开销上不如表分区。总之，上述三种优化方式（索引、表分区、物化视图）在Q1上各有优劣，但均起到了一定的优化效果，之后会在更多query上进一步探讨。

**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。





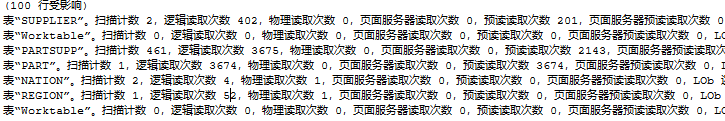
2.2、索引优化分析

【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。

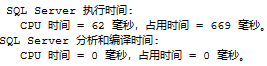


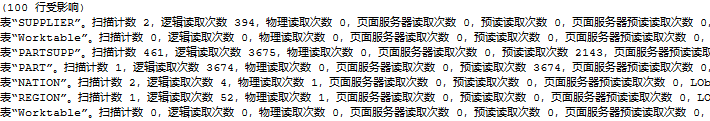


【优化2】更进一步地，我们还可以建立更多索引。然而，其他需要添加索引的列均为KEY列，而PART、SUPPLIER等表的KEY都建立了聚簇索引，因此实际上再建立非聚集索引意义不大，作为实验，建立如下索引，再执行查询，结果如下。

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);



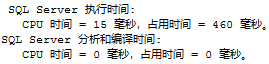


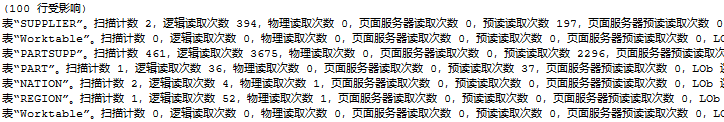
可见，运行时间略有减少，读盘次数也有小幅度减小，但建立索引本身也有时间、空间代价，这种小幅度优化实际上意义不大。

【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。

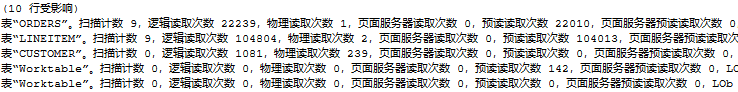




总之，本查询的执行时间从2700ms优化到460ms，效果明显。同时，从中也不难发现：①不是加索引就好，有时候加索引开销大，同时也没有什么性能提升；②在多个列均可加索引时，应该优先选择条件约束高、索引key占用空间小、条件判定速度快的（例如int的=判定），这样访问索引表时效率高。

**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。

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。

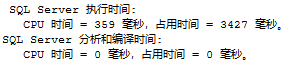
与之前的优化类似，在L\_SHIPDATE、O\_ORDERDATE、C\_MKTSEGMENT上添加索引，同时包含查询可能涉及的其他位域，如下所示。

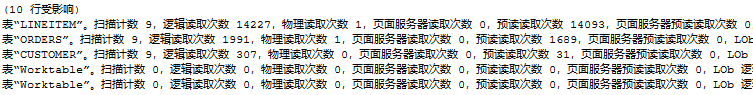
create nonclustered index ix2 on LINEITEM(L\_SHIPDATE) include (L\_EXTENDEDPRICE, L\_DISCOUNT);

create nonclustered index ix1 on ORDERS(O\_ORDERDATE) include (O\_CUSTKEY, O\_SHIPPRIORITY);

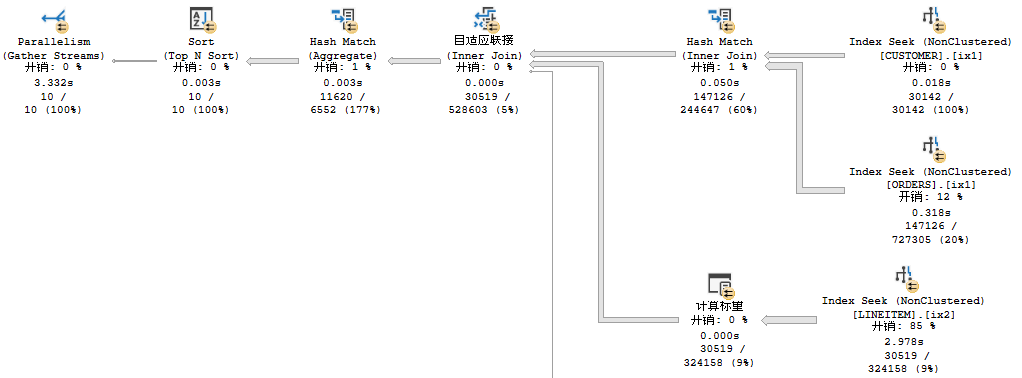
create nonclustered index ix1 on CUSTOMER(C\_MKTSEGMENT);

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。





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

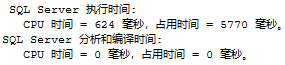
AS PARTITION tpch\_partition\_q3char TO (GROUP1, GROUP2, GROUP3, GROUP4, GROUP5 );

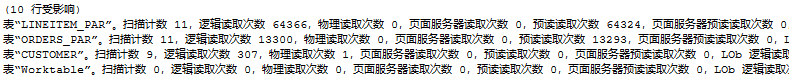
这里q3date应用于ORDERS和LINEITEM表，基于date数据类型分区，q3char应用于CUSTOMERS表，利用char(10)字符串进行分区。

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。





【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

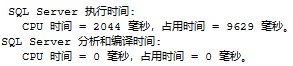
RANGE LEFT FOR VALUES (-1073741824, 0, 1073741824)

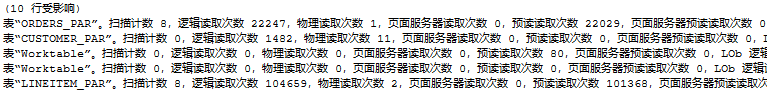
drop partition scheme sche\_fun\_hash

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。





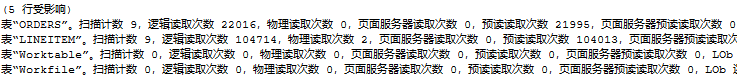
可见，表分区并非万能的，需要根据查询在某些可以迅速缩减扫描范围的列上将表分区，方可达到效果，方案一就是一个很好的展示。

4、Query 4

4.1、基准结果

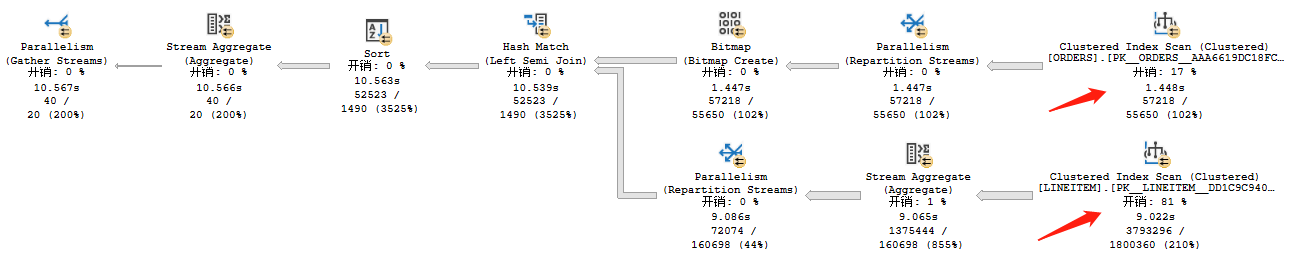
在没有任何索引、分区等的情况下运行查询，结果如下。经验证，答案正确无误。





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。

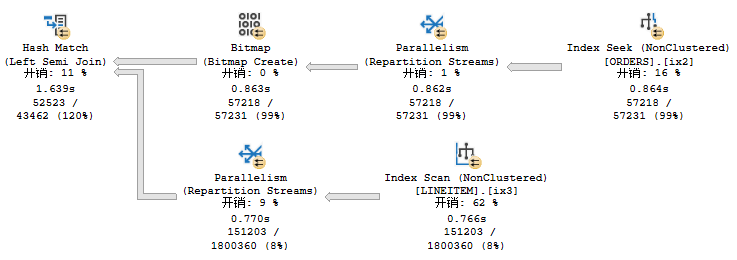


与之前方法相同，在查询的where字句中涉及的列上添加非聚集索引，同时将其他涉及的列加入覆盖索引中。具体在本查询中，应该分别以O\_ORDERDATE、L\_ORDERKEY为键建立索引，如下。

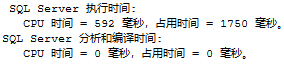
create nonclustered index ix2 on ORDERS(O\_ORDERDATE) include (O\_ORDERPRIORITY);

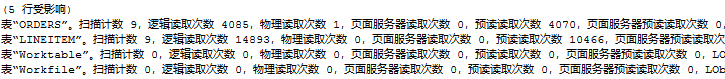
create nonclustered index ix3 on LINEITEM(L\_ORDERKEY) include (L\_COMMITDATE, L\_RECEIPTDATE);

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。





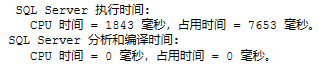
4.3、表分区优化分析

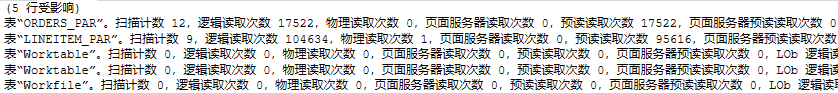
【方案1】

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**。

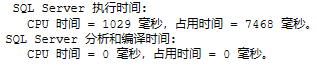


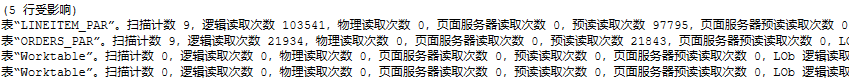


这一结果固然说明表分区有效，但一方面效果远不如索引，另一方面也没有发挥表分区并行化的优势，因此可以尝试其他分区方案。

【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。



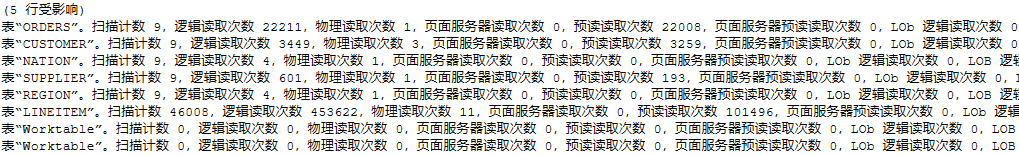
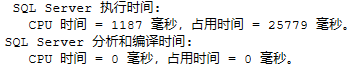


可见，IO次数略有增多，占用时间略有减少。这较为符合预期，说明此时的分区主要依靠并行化提高性能。但是可以注意到，此处的变化较小，实际上难以排除是偶然因素、系统自身因素导致的结果，所以我们无法100%下定结论。但不论如何，可以发现对于Q4表分区带来的优化明显不如索引，这说明在不同问题上表分区、索引效果不同，需要通过实践得到更好的选择。

**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。



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)

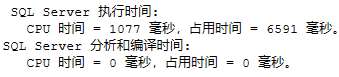
include (L\_ORDERKEY, L\_EXTENDEDPRICE, L\_DISCOUNT);

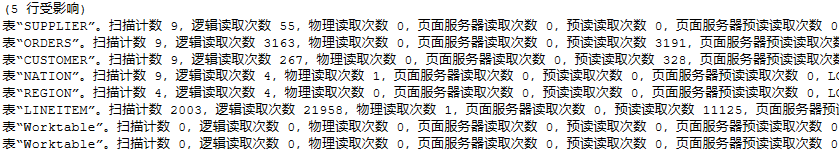
create nonclustered index ix3 on SUPPLIER(S\_NATIONKEY);

create nonclustered index ix3 on NATION(N\_REGIONKEY);

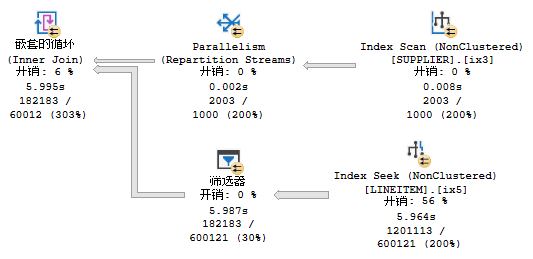
create nonclustered index ix2 on REGION(R\_NAME);

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。





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)

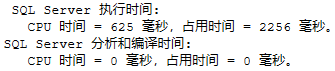
include (L\_SUPPKEY, L\_EXTENDEDPRICE, L\_DISCOUNT);

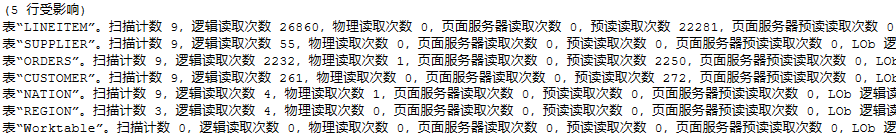
create nonclustered index ix2 on SUPPLIER(S\_SUPPKEY) include (S\_NATIONKEY);

create nonclustered index ix2 on NATION(N\_NATIONKEY) include (N\_REGIONKEY);

create nonclustered index ix1 on REGION(R\_REGIONKEY) include (R\_NAME);

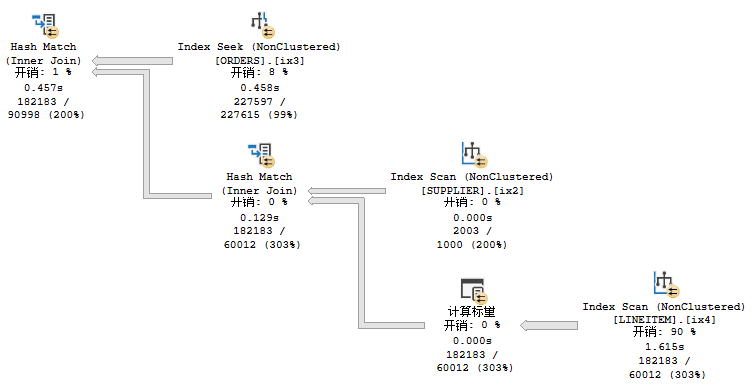
此时执行查询的性能如下所示。可见，此次的优化程度幅度更大，查询时间比Baseline快了约10倍。





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。



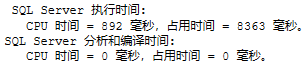
5.3、表分区优化分析

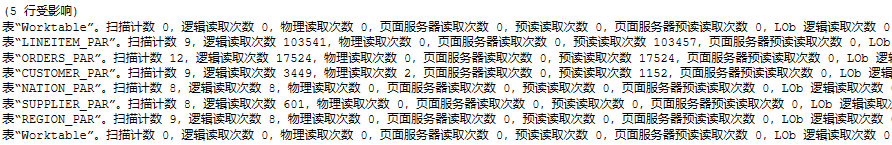
借鉴在索引优化中的结果，按照表的连接顺序从ORDERS向REGION方向建立索引效率更高；因此，在表分区时，也可以优先依据这些key来对表分区。具体地，分区依据如下表所示，其中涉及到的两种分区方案中，tpch\_partition\_scheme\_q3date用于date类型数据的均匀分区，sche\_fun\_hash为哈希分区，应用于int型数据的均匀分区；二者在之前的Query（Q3）已经详述，不再赘述。

按照这一方式将所有表进行分区，详细代码见q5.sql。

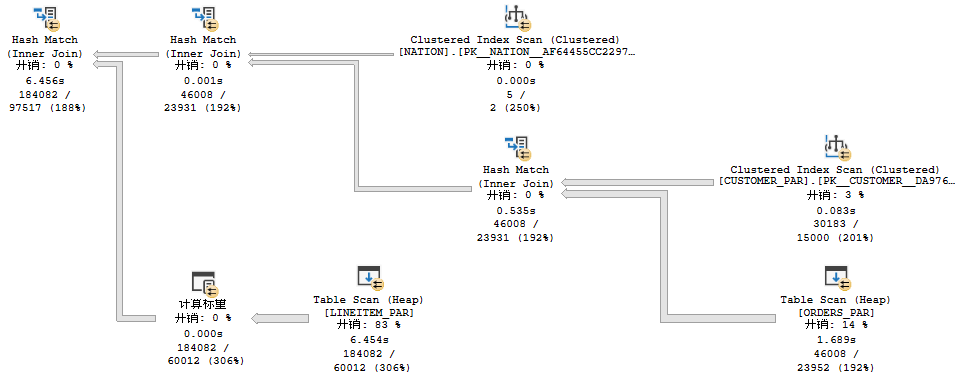
|  |  |  |
| --- | --- | --- |
| 表名 | 分区键 | 分区方案 |
| ORDERS | O\_ORDERDATE | tpch\_partition\_scheme\_q3date |
| LINEITEM | L\_ORDERKEY | sche\_fun\_hash |
| CUSTOMER | C\_CUSTKEY | sche\_fun\_hash |
| SUPPLIER | S\_SUPPKEY | sche\_fun\_hash |
| NATION | N\_NATIONKEY | sche\_fun\_hash |
| REGION | R\_REGIONKEY | 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。





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

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。