综合实习4 性能调优

罗昊 1700010686 常辰 1500012710 易士程 1500012789

任务一 索引调优

数据集

在这一任务中使用来自于MovieLen网站上电影及其评价的数据集ml-latest，该数据集大约有4万多部电影和一共2600万条评论。

表结构

该数据集共有6个表，本次任务只用到其中5个表，第一个表movies包含三列：

- movieId：标识电影的ID
- title：电影名
- genres：电影类型，可以有多种类型

第二个表links包含三列：

- movieId
- imdbId：在imdb中电影的标识
- tmdbId：在themoviedb中电影的标识

第三个表ratings包含4列：

- userId：标识用户的ID
- movieId
- rating：该用户对电影的评分，范围为0.5~5，每0.5为一档
- timestamp：评分的时间戳

第四个表tags包含4列：

- userId
- movieId
- tag：用户给电影打的标签
- timestamp

第五个表genome-scores包含3列：

- movieId
- tagId：标识标签的ID
- releavance：标签与电影的相关度，取值为0~1

准备工作

显示统计信息并清空缓存（每次测试前）

```
set statistics IO on
set statistics time on
dbcc dropcleanbuffers
dbcc freeproccache
```

表连接

按照userId和movieId对表ratings和tags连接，并且不选择timestamp进入最终结果：

```
select R.userId, R.movieId, R.rating, T.tag
from dbo.ratings R inner join dbo.tags T
on R.movieId = T.movieId and R.userId = T.userId
```

索引选择分别覆盖rating和tag的覆盖索引：

```
create index clu_in_ratings
on dbo.ratings(movieId, userId) include(rating)
create index clu_in_tags
on dbo.tags(movieId, userId) include(tag)
```

由此可以得到两者连接的开销

Hash Match

使用来自顶部输入的每一行生成哈希表，使用来自底部输入的每一行探测该哈希表，然后输出所有匹配的行。

物理运算	Hash Match
逻辑操作	Inner Join
实际执行模式	Row
估计的执行模式	Row
Actual Number of Rows	654783
Actual Number of Batches	0
估计运算符开销	5534.13642 (98%)
估计 I/O 开销	3320.07
估计 CPU 开销	2214.06
估计子树大小	5665.34
Number of Executions	1
估计执行次数	1
估计行数	754817
估计行大小	4051 字节
Actual Rebinds	0
Actual Rewinds	0
节点 ID	0

输出列表
[MovieLen].[dbo].[ratings].userId, [MovieLen].[dbo].[ratings].movieId, [MovieLen].[dbo].[ratings].rating, [MovieLen].[dbo].[tags].tag

Hash Keys Probe
[MovieLen].[dbo].[ratings].movieId, [MovieLen].[dbo].[ratings].userId

Probe Residual
[MovieLen].[dbo].[ratings].[movieId] as [R].[movieId]=[MovieLen].[dbo].[tags].[movieId] as [T].[movieId] AND [MovieLen].[dbo].[ratings].[userId] as [R].[userId]=[MovieLen].[dbo].[tags].[userId] as [T].[userId]

Merge Join

从两个已进行了相应排序的输入表中，使用其排序顺序对行进行匹配。

物理运算	Merge Join
逻辑操作	Inner Join
实际执行模式	Row
估计的执行模式	Row
Actual Number of Rows	654783
Actual Number of Batches	0
估计运算符开销	145.78723 (58%)
估计 I/O 开销	78.1085
估计 CPU 开销	67.6788
估计子树大小	250.211
Number of Executions	1
估计执行次数	1
估计行数	753170
估计行大小	4051 字节
Actual Rebinds	0
Actual Rewinds	0
Many to Many	True
节点 ID	0

Where (联接列)
([MovieLen].[dbo].[ratings].movieId, [MovieLen].[dbo].[ratings].userId) = ([MovieLen].[dbo].[tags].movieId, [MovieLen].[dbo].[tags].userId)

输出列表
[MovieLen].[dbo].[ratings].userId, [MovieLen].[dbo].[ratings].movieId, [MovieLen].[dbo].[ratings].rating, [MovieLen].[dbo].[tags].tag

左图不使用索引，因此使用表连接时选择散列连接，右图使用了索引，需要连接的部分是有序的，所以用归并连接。可以看出左边的开销远大于右边，再比较两边的I/O情况和消耗时间：

	扫描次数	逻辑读取次数	预读次数	CPU时间	总时间
不用索引	2	137353	137353	8625ms	16566ms
使用索引	2	**101519**	**61493**	**4047ms**	**14798ms**

使用索引可以大幅减少预读次数和CPU时间，并减少一部分逻辑读取次数和总时间，而创建两个索引的总时间为28044ms，在需要频繁用到连接时使用索引毫无疑问是十分有效的。

分组

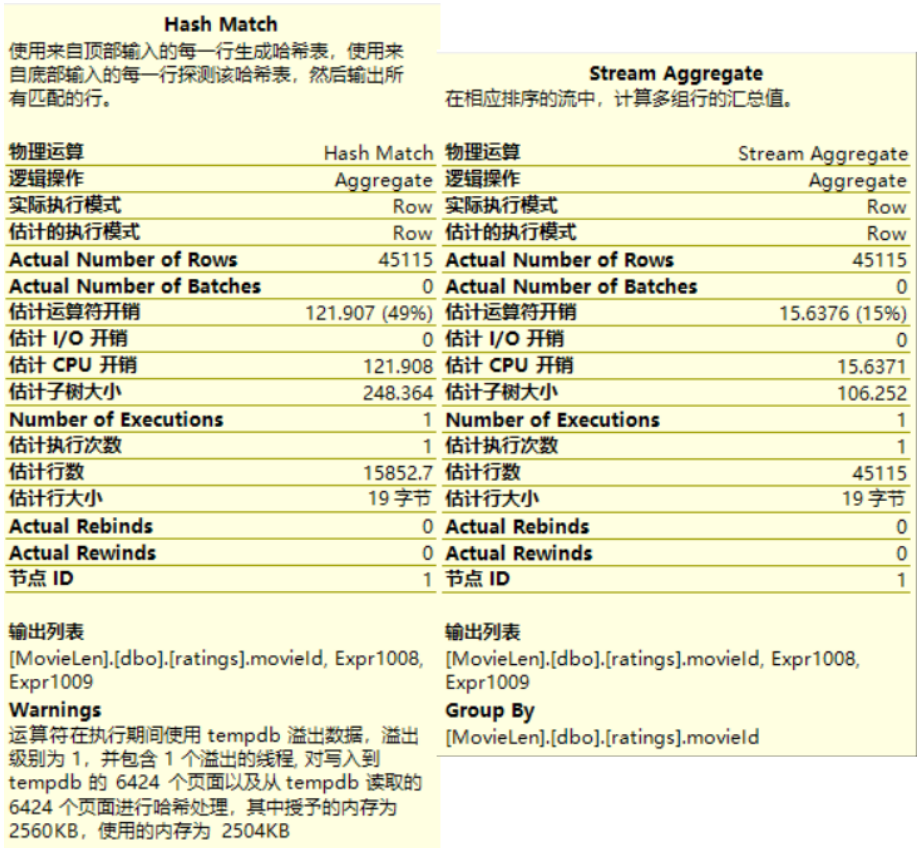
通过对表ratings按照列movieId分组求每个电影的平均得分：

```
select movieId, avg(rating)
from dbo.ratings
group by movieId
```

索引同样选择覆盖索引：

```
create index clu_in_ratings_avg
on dbo.ratings(movieId) include (rating)
```

由此可以得到两者分组的开销



Hash Match
使用来自顶部输入的每一行生成哈希表，使用来自底部输入的每一行探测该哈希表，然后输出所有匹配的行。

物理运算	Hash Match
逻辑操作	Aggregate
实际执行模式	Row
估计的执行模式	Row
Actual Number of Rows	45115
Actual Number of Batches	0
估计运算符开销	121.907 (49%)
估计 I/O 开销	0
估计 CPU 开销	121.908
估计子树大小	248.364
Number of Executions	1
估计执行次数	1
估计行数	15852.7
估计行大小	19 字节
Actual Rebinds	0
Actual Rewinds	0
节点 ID	1

输出列表
[MovieLen].[dbo].[ratings].movieId, Expr1008, Expr1009

Warnings
运算符在执行期间使用 tempdb 溢出数据，溢出级别为 1，并包含 1 个溢出的线程，对写入到 tempdb 的 6424 个页面以及从 tempdb 读取的 6424 个页面进行哈希处理，其中授予的内存为 2560KB，使用的内存为 2504KB

Stream Aggregate
在相应排序的流中，计算多组行的汇总值。

物理运算	Stream Aggregate
逻辑操作	Aggregate
实际执行模式	Row
估计的执行模式	Row
Actual Number of Rows	45115
Actual Number of Batches	0
估计运算符开销	15.6376 (15%)
估计 I/O 开销	0
估计 CPU 开销	15.6371
估计子树大小	106.252
Number of Executions	1
估计执行次数	1
估计行数	45115
估计行大小	19 字节
Actual Rebinds	0
Actual Rewinds	0
节点 ID	1

输出列表
[MovieLen].[dbo].[ratings].movieId, Expr1008, Expr1009

Group By
[MovieLen].[dbo].[ratings].movieId

左图不使用索引，在分组过程中产生了数据溢出，其开销也远大于右边，右边由于得到的数据是有序的，可以通过流合并得到结果，不需要全部两两比较。从I/O情况和总消耗时间也可以看出使用索引的优势：

	扫描次数	逻辑读取次数	预读次数	CPU时间	总时间
不用索引	3	138491	137692	5906ms	7581ms
使用索引	**1**	**83900**	**64201**	**4578ms**	**5292ms**

不使用索引时需要创建临时表储存中间结果，因此除了读取次数和时间之外，还增加了扫描次数。

not exists子查询

MovieLen中查询在themoviedb中ID不小于10000的标题名：

```
select title
from dbo.movies M
where not exists(
    select *
    from dbo.links L
    where M.movieId = L.movieId
          and L.tmdbId < 10000
)
```

只在表links中建立覆盖索引：

```
create index clu_in_links
on dbo.links(tmdbId) include(movieId)
```

得到两者开销

Hash Match 使用来自顶部输入的每一行生成哈希表，使用来自底部输入的每一行探测该哈希表，然后输出所有匹配的行。	
物理运算	Hash Match
逻辑操作	Right Anti Semi Join
实际执行模式	Row
估计的执行模式	Row
Actual Number of Rows	42055
Actual Number of Batches	0
估计运算符开销	0.318723 (34%)
估计 I/O 开销	0
估计 CPU 开销	0.296715
估计子树大小	0.940956
Number of Executions	1
估计执行次数	1
估计行数	42045.5
估计行大小	4035 字节
Actual Rebinds	0
Actual Rewinds	0
节点 ID	0
输出列表	[MovieLen].[dbo].[movies].title
Hash Keys Probe	[MovieLen].[dbo].[movies].movieId

Hash Match 使用来自顶部输入的每一行生成哈希表，使用来自底部输入的每一行探测该哈希表，然后输出所有匹配的行。	
物理运算	Hash Match
逻辑操作	Right Anti Semi Join
实际执行模式	Row
估计的执行模式	Row
Actual Number of Rows	42055
Actual Number of Batches	0
估计运算符开销	0.2967173 (37%)
估计 I/O 开销	0
估计 CPU 开销	0.296715
估计子树大小	0.79196
Number of Executions	1
估计执行次数	1
估计行数	42045.5
估计行大小	4035 字节
Actual Rebinds	0
Actual Rewinds	0
节点 ID	0
输出列表	[MovieLen].[dbo].[movies].title
Hash Keys Probe	[MovieLen].[dbo].[movies].movieId

两边仅在运算符开销和子树大小上有区别，并且效果不明显，不过从I/O情况和消耗时间能体现出索引的优势：

	扫描次数	逻辑读取次数	预读次数	CPU时间	总时间
不用索引	2	697	697	62ms	1052ms
使用索引	2	**590**	**588**	**16ms**	**1046ms**

使用索引在总时间上没有优势，但是节省大量CPU时间和I/O次数，由于索引是作用在表links上，因此links相对于movies数据量较小的情况下体现不出总时间的优势。

单列索引与覆盖索引

在建立索引时将查询时需要的列加入覆盖索引中，可以相比单列索引减少I/O次数。同样求每个电影的平均得分：

```
select movieId, avg(rating)
from dbo.ratings
group by movieId
```

单列索引取movieId这一列，覆盖索引覆盖rating：

```
create index clu_in_ratings_single
on dbo.ratings(movieId)

create index clu_in_ratings_cover
on dbo.ratings(movieId) include (rating)
```

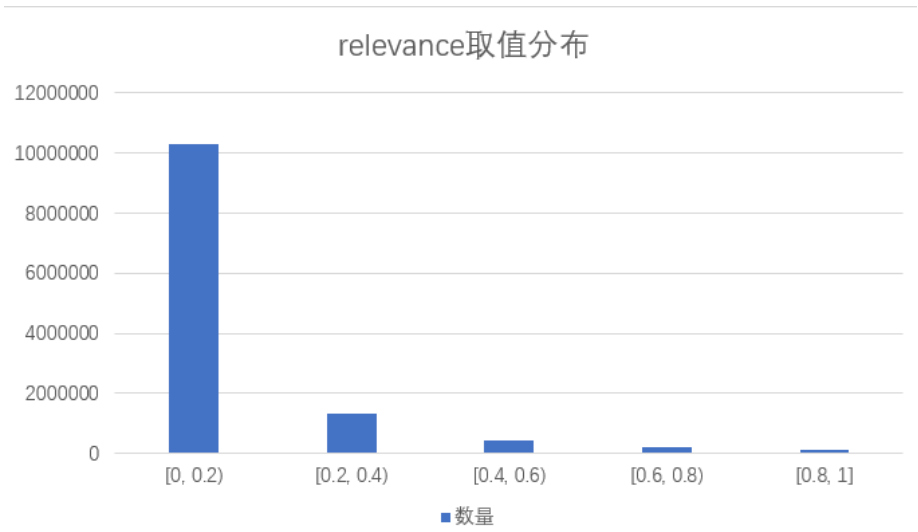
分别进行测试I/O情况：

	扫描次数	逻辑读取次数	预读次数
单列索引	1	132412	132259
覆盖索引	1	**101519**	**61493**

预读次数减少了一半以上，符合预期，实际消耗时间也略小。

过滤索引

首先简单分析表genome-scores中relevance的分布律：



可见在[0.4, 1]区间内分布极其稀疏，建立过滤索引：

```
create index clu_in_score
on dbo.[genome-scores](relevance) include(movieId, tagId)
where relevance >= 0.4
```

同时也建立完整的索引：

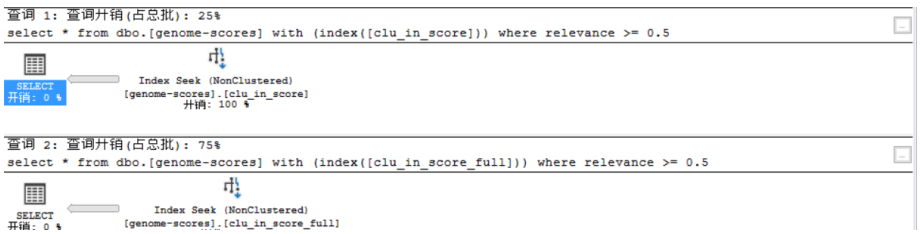
```
create index clu_in_score_full
on dbo.[genome-scores](relevance) include(movieId, tagId)
```

由于现实中大多选取相关度较高的tag查询，因此以相关度不小于0.5的查询作为例子：

```
select *
from dbo.[genome-scores] with (index([clu_in_score]))
where relevance >= 0.5

select *
from dbo.[genome-scores] with (index([clu_in_score_full]))
where relevance >= 0.5
```

于是得到两次查询的开销占比：



采用过滤索引的开销是完整索引的1/3，效果显著。

总结

索引在大部分情况都能大幅降低查询开销，但是由于在大型数据集上建立索引也需要消耗不少时间，因此需要选择合适的索引。

根据以上实验可以得到一些选择的标准，对于经常被查询的列，可以将其加入覆盖索引的覆盖范围。对于分布很不均匀的列，在稀疏的部分建立过滤索引可以进一步降低开销。

任务二 最大并发间隔不同解决方案

这一任务的目标是求出每个用户处于最大并发数的时间段，某个时刻的并发数为"大于等于进程的开始时间，小于结束时间"的进程数。有三种不同方法可以求出，分别为基于集合、游标、窗口函数的解决方案。

表结构

只需要用到一个表Sessions，包含4列：

- id：进程的唯一标识
- username：用户名
- starttime：开始时间
- endtime：结束时间

基于集合的解决方案

这一解决方案的思路如下：

1. 求出用户每个进程开始时间点和结束时间点的并发数
2. 求出每个用户的最大并发数
3. 找到同一用户所有处于最大并发数的开始时间点和结束时间点
4. 对于上述每个开始时间点，找到最近的结束时间点，这一时间段为一个最大并发间隔

下面说明这一方案的正确性，每次并发数发生变化的时刻一定时每个进程的开始时刻或者结束时刻，因此求最大并发数转为一个离散的问题，可以通过取开始时刻并发数的最大值得到。

又由于相邻时刻并发数变化绝对值总是1，所以在所有处于最大并发数的时刻中，取距离开始时刻最近的结束时刻组成的时间段一定是所求时间段。

具体代码如下：

```
WITH TimePoints AS
(
  SELECT username, starttime AS ts FROM dbo.Sessions
),
Counts AS
(
  SELECT username, ts,
    (SELECT COUNT(*)
     FROM dbo.Sessions AS S
     WHERE P.username = S.username
       AND P.ts >= S.starttime
       AND P.ts < S.endtime) AS concurrent
  FROM TimePoints AS P
),
ETimePoints AS
(
  SELECT username, endtime AS ets FROM dbo.Sessions
),
ECounts AS
(
  SELECT username, ets,
    (SELECT COUNT(*)
     FROM dbo.Sessions AS S
     WHERE P.username = S.username
```

```
        AND P.ets > S.starttime
        AND P.ets <= S.endtime) AS concurrent
  FROM ETimePoints AS P
),
Maxsess AS
(
    SELECT username, MAX(concurrent) AS mx
    FROM Counts
    GROUP BY username
),
Maxsts AS
(
    SELECT S.username, ts
    FROM Counts S, Maxsess M
    WHERE S.username = M.username AND S.concurrent = M.mx
),
Maxets AS
(
    SELECT E.username, ets
    FROM ECounts E, Maxsess M
    WHERE E.username = M.username AND E.concurrent = M.mx
)
SELECT distinct username, ts AS starttime,
    (
    SELECT MIN(ets) FROM Maxets AS E
    WHEREe E.username = S.username
        AND ets > ts) AS endtime
FROM Maxsts AS S
```

基于游标的解决方案

这一方案不直接在表Session中查询，而使用重构Session的临时表：

```
SELECT username, starttime AS ts, +1 AS type
  FROM dbo.Sessions

  UNION ALL

  SELECT username, endtime, -1
  FROM dbo.Sessions

  ORDER BY username, ts, type;
```

这个临时表的最后一列type每表示时刻的类型，每个开始时刻并行数+1，每个结束时刻并行数-1，便于计算。方案如下：

1. 在临时表中从上至下扫描

2. 扫描一行，如果用户名发生变化，则初始化变量
3. 将type值加到变量concurrent中，这一变量表示当前时刻的并行数
4. 若超过目前最大并行数，将该用户之前的所有最大并发间隔记录从表usernamesMx中删除，并记录当前时刻
5. 若正好达到目前最大并行数，记录当前时刻
6. 若小于当前最大并行数，判断是否是最大并行间隔的结束时刻，是的话将这一间隔插入表usernamesMx中
7. 全部扫描一遍后算法结束

这一算法结束后，表usernamesMx中保存了所有最大并发间隔。

具体代码如下：

```
DECLARE
  @username AS varchar(10),
  @prevusername AS varchar (10),
  @ts AS datetime,
  @tmps AS datetime,
  @type AS int,
  @concurrent AS int,
  @flag AS int,
  @mx AS int;

DECLARE @usernamesMx TABLE
(
  username varchar (10) NOT NULL,
  starttime datetime NOT NULL,
  endtime datetime NOT NULL
);

DECLARE sessions_cur CURSOR FAST_FORWARD FOR
  SELECT username, starttime AS ts, +1 AS type
  FROM dbo.Sessions

  UNION ALL

  SELECT username, endtime, -1
  FROM dbo.Sessions

  ORDER BY username, ts, type;

OPEN sessions_cur;

FETCH NEXT FROM sessions_cur
  INTO @username, @ts, @type;

SET @prevusername = @username;
SET @concurrent = 0;
SET @mx = 0;
```

```
SET @flag = 0;

WHILE @@FETCH_STATUS = 0
BEGIN
  IF @username <> @prevusername
  BEGIN
    SET @concurrent = 0;
    SET @mx = 0;
    SET @flag = 0;
    SET @prevusername = @username;
  END

  SET @concurrent = @concurrent + @type;
  IF @concurrent > @mx
  BEGIN
    SET @mx = @concurrent;
    DELETE FROM @usernamesMx WHERE username = @username
    SET @flag = 1;
    SET @tmps = @ts;
  END
  ELSE IF @concurrent = @mx
  BEGIN
    SET @flag = 1;
    SET @tmps = @ts;
  END
  ELSE IF @flag = 1
  BEGIN
    SET @flag = 0;
    INSERT INTO @usernamesMx VALUES(@username, @tmps,
@ts);
  END

  FETCH NEXT FROM sessions_cur
    INTO @username, @ts, @type;
END

CLOSE sessions_cur;

DEALLOCATE sessions_cur;

SELECT * FROM @usernamesMx;
GO
```

基于窗口函数的解决方案

这一方案仍然需要利用上一小节中的临时表，思路如下：

1. 建立上述临时表

2. 利用窗口函数对每个用户分别累加type值，记为cnt，即cnt为该用户的当前和之前的行所有type值之和
3. 找到每个用户的cnt最大值的行，这一行的时刻为最大并行数的起始时刻
4. 找到每个用户的cnt最大值 - 1，且type为-1的行，这一行是最大并行数的结束时刻
5. 将3，4中得到的表并排连接起来

具体代码如下：

```
WITH C1 AS
(
  SELECT username, starttime AS ts, +1 AS type
  FROM dbo.Sessions

  UNION ALL

  SELECT username, endtime, -1
  FROM dbo.Sessions
),
C2 AS
(
  SELECT *,
  SUM(type) OVER(PARTITION BY username ORDER BY ts, type
                ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW) AS cnt
  FROM C1
),
ET AS
(
  SELECT A.username, A.ts AS endtime,
    ROW_NUMBER() OVER(PARTITION BY username ORDER BY ts)
AS rownum FROM C2 A
  WHERE cnt = (SELECT MAX(cnt) FROM C2 B WHERE A.username
= B.username AND A.type = -1) - 1
),
ST AS
(
  SELECT A.username, A.ts AS starttime,
    ROW_NUMBER() OVER(PARTITION BY username ORDER BY ts)
AS rownum FROM C2 A
  WHERE cnt = (SELECT MAX(cnt) FROM C2 B WHERE A.username
= B.username)
)
SELECT S.username, S.starttime, E.endtime
FROM ET E, ST S
WHERE E.username = S.username AND E.rownum = S.rownum
```

三种方法的比较分析

数据量较小的情况（16行）：

	扫描次数	逻辑读取次数	CPU时间	总时间
基于集合	150	325	16ms	40ms
基于游标	-	-	-	4281ms
基于窗口函数	10	122	0ms	68ms

基于游标的方案难以总计扫描次数等信息，只给出总时间，游标的执行计划中，对结果表的频繁修改是导致运行时间过长的原因。

基于窗口函数的I/O开销明显优于基于集合的解决方案，仅在总时间上略有劣势，这说明在较小的数据集上直接采用最普通的基于集合的方法也是一种比较不错的选择。

接下来看中型数据量的情况（1万行）：

	扫描次数	逻辑读取次数	预读次数	CPU时间	总时间
基于集合	584648	4611758	0	58141ms	58578ms
基于窗口函数	8	4488	1438	3171ms	3484ms

由于基于游标的方法开销过大，已经不适用与这一情况，因此不考虑这一方法。

最后看数据量很大的情况（500万行）：

	扫描次数	逻辑读取次数	预读次数	CPU时间	总时间
基于集合	-	-	-	-	>5h
基于窗口函数	8	224304	99722	181641ms	183621ms

跟据以上三种不同的数据量，可以推断出基于窗口函数的方法是最优的方法，而基于集合的方法在小数据量的时候仍能适用，但是基于游标的方法和其他方法差距较大，难以适用。

任务三 Skyline的有效计算

数据集

使用DBGen生成两种维度下各300万条数据

表结构

该数据集有两个表，第一个表Hotel2D包含3列：

- id：标识数据的ID
- distance：酒店与目的地距离，取值为0~30000，越小越好

- price：价格，取值为100~10000，越低越好

第二个表Hotel3D包含4列：

- id
- distance
- price
- rating：酒店评价，取值为1~10，越高越好

原始查询

数据为二维时使用的查询，结果集大小为8：

```
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
)
```

数据为三维时使用的查询，结果集大小为85：

```
select  *
from    dbo.Hotel3D  h
where   not exists
(    select *
    from    dbo.Hotel3D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   h1.rating >=  h.rating
      and   ( h1.distance < h.distance or h1.price <
h.price or h1.rating >  h.rating)
)
```

结果集大小明显随维度增加而大幅增加，考虑数据均匀分布的情况，二维结果数量可以看作是矩形两条边长的和，三维结果的数量可以看作是三个矩形的面积和，有不少差距。

分治算法

将集合分为10个（二维数据），分别计算结果再合并计算结果。

具体代码如下：

```
with sub1 as (
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 0 and 300000)
) and (h.id between 0 and 300000)

union all

select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 300000 and 600000)
) and (h.id between 300000 and 600000)
union all
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 600000 and 900000)
) and (h.id between 600000 and 900000)
union all
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
```

```
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 900000 and 1200000)
) and (h.id between 900000 and 1200000)
union all
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 1200000 and 1500000)
) and (h.id between 1200000 and 1500000)
union all
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 1500000 and 1800000)
) and (h.id between 1500000 and 1800000)
union all
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 1800000 and 2100000)
) and (h.id between 1800000 and 2100000)
union all
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
```

```
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 2100000 and 2400000)
) and (h.id between 2100000 and 2400000)
union all
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 2400000 and 2700000)
) and (h.id between 2400000 and 2700000)
union all
select  *
from    dbo.Hotel2D  h
where   not exists
(    select *
    from    dbo.Hotel2D h1
    where   h1.distance <= h.distance
      and   h1.price <= h.price
      and   ( h1.distance < h.distance or h1.price <
h.price )
      and   (h1.id between 2700000 and 3000000)
) and (h.id between 2700000 and 3000000)
)
select * from sub1 s
where not exists (
    select * from sub1 s1
    where s1.distance <= s.distance
      and   s1.price <= s.price
      and   ( s1.distance < s.distance or s1.price <
s.price )
)
```

性能结果：

	扫描次数	逻辑读取次数	预读次数	CPU时间	总时间
原始查询	3	18777101	9320	60516ms	60784ms
基于窗口函数	61	34561507	9320	98031ms	98677ms

分治算法相比原始查询并没有提升性能，反而下降了。这可能是由于在分治时选取不同的集合需要扫描全表。

建立索引

在id之外的所有列建立索引：

```
create index in_2d
on dbo.Hotel2D(distance, price)

create index in_3d
on dbo.Hotel3D(distance, price, rating)
```

再进行查询，得到性能结果：

	扫描次数	逻辑读取次数	预读次数	CPU时间	总时间
无索引2D	3	18777101	9320	60516ms	60784ms
有索引2D	3000001	12036200	8177	19156ms	19440ms
无索引3D	3	25315999	12288	324469ms	325235ms
有索引3D	3000001	12788731	11197	30407ms	30610ms

读取次数较无索引时略为减少，而CPU时间和总时间大幅缩减，而且维度越大，缩减幅度越大，二维时变为约1/3，三维时变为约1/10，当数据维度较大时，虽然建立索引需要消耗较多时间，但是相比查询时的提升是微乎其微的。