4.3. 性能优化

源码部分：optimized_query1_index.sql，optimized_query2_cover index.sql，
optimized_query3&4_materialized view.sql，
optimized_query3&4_redundancy column.

参与者与分工：

- 易港： 查询优化与分析

这个任务主要是针对上一部分的查询进行性能上的优化。

4.3.1 优化指标

通过在同一环境下查询执行时间来判断是否优化,而查看 IO 信息有助于我们分析优化原因。

主要代码：

```
SET STATISTICS TIME ON
SET STATISTICS IO ON
QUERY PROCESS
SET STATISTICS TIME OFF
SET STATISTICS IO OFF
```

4.3.2 准备工作

由于缓存可能会对我们的优化判断造成影响,所以我们在每次查询之后都会清除缓存。

主要代码：

```
DBCC FREEPROCCACHE
DBCC FREESESSIONCACHE
DBCC FREESYSTEMCACHE('ALL')
DBCC DROPCLEANBUFFERS
```

4.3.3 优化方法介绍

常见的调优方法主要有：

- 建立索引：因为查询执行的大部分开销是磁盘 I/O，使用索引提高性能的一个主要目标是避免全表扫描，因为全表扫描需要从磁盘上读表的每一个数据页，如果有索引指向数据值，则查询只需读几次磁盘就可以了。所以如果建立了合理的索引，优化器就能利用索引加速数据的查询过程。
- 冗余列：将一个表的列添加到另一个表中，避免表连接操作。
- 物化视图:物化视图可以用于预先计算并保存表连接或聚集等耗时较多的操作的结果，这样，在执行查询时，就可以避免进行这些耗时的操作，而从快速的得到结果。物化视图对应用透明，增加和删除物化视图不会影响应用程序中 SQL 语句的正确性和有效性;物化视图需要占用存储空间;当基表发生变化时，物化视图也应当刷新。
- 按列存储：列存储不同于传统的关系型数据库，其数据在表中是按行存储的，列方式所带来的重要好处之一就是,由于查询中的选择规则是通过列来定义的,因此整个数据库是自动索引化的。按列存储每个字段的数据聚集存储，在查询只需要少数几个字段的时候，能大大减少读取的数据量，一个字段的数据聚集存储，那就更容易为这种聚集存储设计更好的压缩/解压算法。

4.3.4 优化查询

4.3.4.1 使用索引和按列存储优化 Query1:统计每个 genres 下的电影数量

由于在这个查询中需要对 genres 进行分组，所以我们考虑对 genres 建立非聚簇索引。

```
CREATE NONCLUSTERED
INDEX GENRES_NONCLU_INDEX
ON dbo.MovieType(genres)
```

我们需要查询的信息全部包含在 genres 列中，所以我们对 genres 建立列存储索引。

```
CREATE NONCLUSTERED COLUMNSTORE
INDEX GENRES_NONCLU_COL_INDEX
ON dbo.MovieType(genres)
```

优化结果如下：

表 2：Query1 的优化结果

	不建索引	普通索引	列存储索引
IO 信息	逻辑读取 257 次 物理读取 1 次 预读 255 次 lob 逻辑读取 0 次 lob 物理读取 0 次 lob 预读 0 次	逻辑读取 224 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。	逻辑读取 0 次 物理读取 0 次 预读 0 次 lob 逻辑读取 72 次 lob 物理读取 1 次 lob 预读 140 次
执行时间	219ms	92ms	7ms

优化分析：建立普通索引可以减少磁盘 IO 次数，从而减少查询时间，列存储索引并没有在磁盘上读取而是在 lob 上读取。在 lob 上读取的是对象的内容，而在磁盘上读取的是页的内容，这里也可以看出行存储和列存储的不同。

4.3.3.2 使用覆盖索引优化 Query2:统计每个 genres 下电影的平均用户评分

这个查询需要将两个表进行连接，并对 genres 进行分组。首先可以对连接项 MovieId 建立索引，同时观察到我们查询的内容还包括用户的评分，所以考虑建立覆盖索引。我们可以通过实验结果比较两者的区别。

建立覆盖索引代码：

```
CREATE NONCLUSTERED
INDEX MovieID_INCLUDE_INDEX
ON dbo.Ratings(MovieId)
INCLUDE (Rate)
```

优化结果如下：

表 3：Query2 的优化结果

	不建索引	普通索引	覆盖索引
IO 信息	表 'MovieType': 逻辑读取 257 次 物理读取 1 次 预读 255 次 表 'Ratings': 逻辑读取 93717 次	表 'MovieType': 逻辑读取 257 次 物理读取 1 次 预读 255 次 表 'Ratings': 逻辑读取 93717 次	表 'MovieType': 逻辑读取 257 次 物理读取 1 次 预读 255 次 表 'Ratings': 逻辑读取 58085 次

	物理读取 3 次 预读 93713 次	物理读取 0 次 预读 93277 次	物理读取 638 次 预读 48165 次
执行时间	27093ms	27531ms	20734ms

优化分析：只对 MovieId 建立索引，由于需要查询评分，所以需要进行回表操作，性能并没有得到明显提升。覆盖索引是 select 的数据列只用从索引中就能够取得，不必读取数据行，换句话说查询列要被所建的索引覆盖。因此不需要回表操作，也就减少了读取次数，从而减少查询时间。

4.3.3.3 使用物化视图优化 Query3&4

这两个查询都是需要统计观看某个或某类电影的人数，并且得出观看人数超过某一阈值的热门电影，所以我们可以对这个子查询建立一个物化视图，从而减少查询时间。

物化视图的建立：

```
CREATE VIEW PopMovie_VIEW
WITH SCHEMABINDING
AS
  (SELECT MovieId
  FROM dbo.PopularMovie)
GO

CREATE UNIQUE CLUSTERED
INDEX MovieID_INDEX ON PopMovie_VIEW(MovieId)
GO
```

表 4 Query3 的优化结果

	原始查询	物化视图
IO 信息	表 'Movie'：逻辑读取 20 次，物理读取 8 次，预读 0 次。 表 'UserMovieMatrix_id'：逻辑读取 76395 次，物理读取 3 次，预读 76370 次。 表 'MovieMsg'：逻辑读取 243 次，物理读取 1 次，预读 241 次。 表 'Ratings'：逻辑读取 93717 次，物理读取 3 次，预读 93713 次。 表 'Tags'：逻辑读取 4093 次，物理读取 0 次，预读 4026 次。	表 'Movie':逻辑读取 20 次，物理读取 8 次，预读 0 次。 表 'UserMovieMatrix_id' 逻辑读取 76395 次，物理读取 3 次，预读 76391 次。 表 'PopularMovie'：逻辑读取 2 次，物理读取 0 次，预读 0 次。
执行时间	62062ms	29875ms

表 5：Query4 的优化结果

	原始查询	物化视图
IO 信息	表 'Movie'：逻辑读取 360 次，物理读取 51 次，预读 0 次。	表 'Movie':逻辑读取 609 次，物理读取 1 次，

	表 'UserMovieMatrix_id'：逻辑读取 76395 次，物理读取 3 次，预读 76391 次。 表 'MovieMsg'：逻辑读取 243 次，物理读取 1 次，预读 241 次。 表 'Ratings'：逻辑读取 93717 次，物理读取 3 次，预读 93713 次。 表 'Tags'：逻辑读取 4093 次，物理读取 0 次，预读 4093 次。	预读 50 次。 表 'UserMovieMatrix_id' 逻辑读取 76395 次，物理读取 3 次，预读 76391 次。 表 'PopularMovie'：逻辑读取 2 次，物理读取 2 次，预读 0 次。
执行时间	62828ms	29859ms

优化分析：物化视图其实只是将子查询的结果保存起来，为以后多个使用到该子查询结果的查询省去了这一步，所以减少了执行时间。

4.3.3.4 使用冗余列优化 Query3&4

由于在这两个查询中需要输出电影的名称，我们考虑将电影名称（title）这一列添加到 UserMovieMatrix_id 这个表中，从而除去一个表连接操作。

对于冗余列，由于需要更新某一个表，这个时间比较长，所以我们只是写了优化代码，但没有执行。

5. 实验结果

5.2 查询优化

优化结果已经展示在 4.3 中。