Lab6 PostgreSQL 查询优化与执行计划初探

本次实验带领大家初步了解 PostgreSQL 是如何对一条 SQL 查询语句进行优化。
由于 PostgreSQL 真正的查询优化器实现原理复杂且工程量很大，因此本次实验内容仅包含查询优化模块最基础的内容：谓词下推、子查询提升、单表扫描、两表连接。
如果想深入了解 PostgreSQ L查询优化原理与实现，可以阅读参考资料中的书籍和文档。

本次实验配合网课视频第六章一起学习更佳

一、何为查询优化

可以想象一个现实中的例子：现在有个任务，要求从华东师范大学（中北校区）到外滩，请问该如何设计通行路线使所用时间最少。可以确定的是，到达外滩的可选路线有无数条，但是我们不可能都走一遍来确定最优路线。我们首先可以从经验入手推导出设计路线的原则，比如尽量使总路程最短、尽量减少步行的距离、尽量乘坐通行速度快的交通工具、尽量减少更换交通工具的次数等，由此可以排除掉大量的路线，得到若干个备选方案（已经远小于初始的问题空间）。接下来我们可以利用真实的数据，比如目前上海市中心车流量堵塞情况、步行/共享单车/地铁等通行方式的速度、地铁的等待时间与换乘时间、红绿灯等待时间等数据，来对不同的备选方案进行时间的估算，最终可以得到优化的通行路线。

关系数据库系统查询优化的原理与上述过程类似，当 RDBMS 接收到一条 SQL 查询语句，需要将 SQL 语句描述的做什么（what）转化为怎么做（how）。这个过程大体可以划分为三个阶段:

1. 将 SQL 语句转化为语法分析树
2. 利用关系代数等价转换进行逻辑优化生成关系代数表达式
3. 利用系统与统计信息对不同的查询路径进行代价估计进行物理优化，最终生成查询执行计划。

其中，第一个阶段本次 Lab 不做探讨，有兴趣的同学可以查找 SQL 语法分析相关的内容，本次 Lab 简要讲解后两个阶段。

二、准备工作（构造实验数据与了解explain命令）

首先构造数据集，在本地 SQL Shell（推荐使用）或者阿里云 PostgreSQL 中执行下述 SQL 命令：

注：如果本地打开 SQL Shell 启动服务失败，可使用下述方式启动服务后再打开 SQL Shell： 打开任务管理器 -> 查看“服务”一栏 -> 找到 "postgresql-x64-14" 服务，如果状态不是正在运行，则右键点击选择“开始”服务，状态显示为“正在运行” -> 从 SQL Shell 中连接数据库（参考lab4）

注：在水杉平台 terminal 中访问阿里云数据库时，每位同学请使用自己的私有数据库,不要创建新的数据库，创建的新数据库为全局共享其他同学也可以访问到，数据不安全。

执行下述命令，删除之前的表数据：

```
drop table spj;
drop table s;
drop table p;
drop table j;
```

创建新的 S, P, J, SPJ 关系表，其中 S.Sno、P.Pno、J.Jno、SPJ.SPJ_ID 属性为主键，存在 B+ 树索引。SPJ 中的 Sno、Pno、Jno 属性为外键属性，没有 B+ 树索引。

```
create table S (Sno int , Sname char(6), City char(4), primary
key(Sno));
create table P (Pno int , Pname char(6), weight int, primary key(Pno));
create table J (Jno int , Jname char(6), City char(4), primary
key(Jno));
create table SPJ ( SPJ_ID int, Sno int, Pno int, Jno int, QTY int,
                   primary key(SPJ_ID), foreign key(Sno) references
S(Sno),
                   foreign key(Pno) references P(Pno), foreign key(Jno)
references J(Jno));
```

生成测试数据：

首先创建一个生成固定字符串长度的函数，用于数据的生成：

```
CREATE OR REPLACE FUNCTION random_string(
  num INTEGER,
  chars TEXT default 'abcdefghijklmnopqrstuvwxyz'
) RETURNS TEXT
LANGUAGE plpgsql
AS $$
DECLARE
  res_str TEXT := '';
BEGIN
  IF num < 1 THEN
      RAISE EXCEPTION 'Invalid length';
  END IF;
  FOR __ IN 1..num LOOP
    res_str := res_str || substr(chars, floor(random() *
length(chars))::int + 1, 1);
  END LOOP;
  RETURN res_str;
END $$;
```

向 S 表中插入1000条记录，P 表插入50条记录，J 表插入5000条记录，SPJ 表插入100000条记录。记录均按照主键大小顺序排列。

`generate_series(n,m)` 表示生成从 n 到 m 的整数序列
`random_string(n)` 表示随机生成长度为 n 的字符串
`random()` 表示随机生成[0, 1]的自然数

```
insert into S select generate_series(1, 1000) as key, random_string(6),
random_string(4);

insert into P select generate_series(1, 50) as key, random_string(6),
```

```
(random()*(10^3))::integer;

insert into J select generate_series(1, 5000) as key, random_string(6),
random_string(4);

insert into SPJ select generate_series(1, 100000) as key, (random()*999
+ 1)::integer, (random()*49 + 1)::integer, (random()*4999 +
1)::integer, (random()*500)::integer;
```

为了观察到一条 SQL 查询具体的执行计划，需要用到 explain 命令。 explain 命令是 PostgreSQL 数据内置的一条命令，命令格式为：

```
explain [option] [SQL查询语句];
```

注：后续的 SQL 命令实验均在本地 PostgreSQL14 版本环境中运行，如果使用不同的版本或者在阿里云环境中运行，结果可能会略有不同。

输入如下命令：

```
testdb=# explain select * from p where p.pno = 1;
                      QUERY PLAN
-----------------------------------------------------
 Seq Scan on p  (cost=0.00..1.63 rows=1 width=15)
   Filter: (pno = 1)
(2 行记录)
```

输出 "select * from p where p.pno = 1;" 这条语句经过 PostgreSQL 查询优化器后生成的执行计划，但并未真正执行，输出的数值均为优化器预测值。输出结果中每个部分的意义会在后面解释。 如果想实际运行 SQL 语句，获得真实值，可以加上 analyze 选项，让 SQL 语句真正运行：

```
testdb=# explain analyze select * from p where p.pno = 1;
                                              QUERY PLAN
------------------------------------------------------------------------------
---------------------
 Seq Scan on p  (cost=0.00..1.63 rows=1 width=15) (actual
time=0.523..0.526 rows=1 loops=1)
   Filter: (pno = 1)
   Rows Removed by Filter: 49
 Planning Time: 0.736 ms
 Execution Time: 0.566 ms
(5 行记录)
```

"actual time" 与 "Execution Time" 是在SQL语句执行过程得到的真实数据。

注：本次实验只涉及到 SQL 查询优化和生成物理执行计划内容，因此不需要使用 analyze 选项实际运行 SQL 查询。如果使用阿里云数据库，为了节省计算资源，只使用 explain 命令即可。

三、逻辑优化之谓词下推

现有一个查询：

```
Select P.Pno, P.Pname
from SPJ, P
where SPJ.Pno = P.Pno and SPJ.QTY >= 500;
```

该查询可以写成一个关系代数表达式：

$$ \pi_{(P.Pno, P.Pname)}(\sigma_{SPJ.QTY >= 500}(SPJ \bowtie_{spj.pno = p.pno} P)) $$

该关系代数表达式表示首先将 SPJ 与 P 两表的数据进行 join 操作生成中间结果，之后筛选出 QTY >= 500 元组，最后将筛选出的元组中 Pno Pname 两列输出。

这个关系代数表达式仅为上述 SQL 查询语义的一种表达方式，实际上还存在许多种表达同一语义的关系代数表达式（有很多条到达外滩的道路）。逻辑优化的目的，就是运用关系运算符与运算规则，对一个初始的关系表达式在保持语义不变的前提下做变换（也称等价变换），降低SQL执行过程中的时间与空间复杂度。

常见的逻辑优化有谓词下推、子查询提升、外链接消除、条件化简等方法。本次lab仅简要简介谓词下推和子查询提升。

首先讲解谓词下推。

选择、投影、连接是 SQL 查询最基础的三种操作，连接操作涉及物理优化，选择、投影操作的逻辑优化原则如下：

- 选择：选择操作本质是对一组元组集合做限定条件，筛选出符合条件的元组。由于选择操作会显著减少元组的数量，因此应将选择操作尽量下推到更低一级的运算中，以减少生成的中间结果，达到优化 I/O 和 CPU 消耗的目的。
- 投影：投影的本质是限制返回元组的列。优化投影操作方式与选择操作类似，尽量下推到更低一级的运算，减少生成的中间元组的大小。与选择操作不同的是，投影操作是消减单个元组的大小，而选择操作是消减元组的数量（一个纵向一个横向）。

上述关系代数表达式中 $\sigma_{SPJ.QTY >= 500}$ 可以选择谓词下推，在 SPJ 与 P 表做连接之前先作用在 SPJ 表上，在不改变语义的前提下，消减 SPJ 与 P 表连接的中间结果数量。根据投影的串接律：如果 ${A1,A2,...,An}$ 是 ${B1,B2,...,Bm}$ 的属性子集，则 $\pi_{A1,A2,...,An}(\pi_{B1,B2,...,Bm}(E)) = \pi_{A1,A2,...,An}(E) $，由于 $\pi_{P.Pno, P.Pname}$ 是 P 表属性的子集，因此可将投影 $\pi_{P.Pno, P.Pname}$ 下推至 P 表，以进一步消减 SPJ 与 P 表连接生成的中间结果大小。

因此，上述关系表达式在应用选择与投影操作谓词下推优化下，可转化为以下语义等价的关系表达式：

$$ \pi_{(P.Pno, P.Pname)}((\sigma_{SPJ.QTY >= 500}(SPJ)) \bowtie_{spj.pno = p.pno} (\pi_{P.Pno, P.Pname}(P))) $$

显然，相较原关系表达式先连接、再选择、最后投影的操作顺序，优化后的关系表达式将选择、投影下推，先对表 SPJ 表做选择操作，筛选出符合条件的元组；再对 P 表做投影减少连接时 P 表的大小，最后进行连接。显著降低了中间结果的大小，提高了 I/O 和 CPU 效率。

四、逻辑优化之子查询提升

子查询在 SQL 语句中是一类常见操作。在早期的数据库实现中，查询优化器会采用嵌套执行的方式，即父查询每处理一行，子查询就调用一次。这样带来的结果是，父查询中执行了多少行，子查询就被调用多少次，导致执行效率低下。

因此，对子查询的优化对 SQL 查询的执行性能会带来很大的影响。

子查询的类型有很多，根据子查询和父查询之间有没有相关性可分为：相关子查询和非相关子查询；根据子查询操作符可划分为：[NOT]IN/ALL/ANY/SOME 子查询与 [NOT]EXISTS 子查询等，不同的子查询类型有不同的优化方式。

本节以in子查询为例，研究子查询优化中的一种关键方法：子查询提升。

```
select *
from J
where J.city in (select city from S);
```

上述 SQL 查询表达的语义是：找出所在地存在工厂（S）的零件厂（J）。

如果按照早期的执行方式执行上述查询，会对 J 表中每一条元组，都执行一遍子查询 `select city from S`，然后判断 J.city 是否在子查询返回的结果中。显然子查询会执行 |J| 遍，造成严重的性能浪费。

现在 SQL 查询优化器会尽量将子查询“提升”，提升到和父查询同一层次。比如上述的 SQL 语句，根据执行的流程可知，子查询每次都在做重复的工作，且目的在于提供 “city” 集合，供父查询中 J 表元组做匹配操作，如果 J.city 在集合中则返回，如果不在集合中则跳过。根据前面的分析可知，in 操作的语义与 join 操作类似，在 join 操作中左连接元组属性值在右连接元组中不存在，则不会连接成连接元组；如果存在，则会生成连接元组，生成连接结果。因此可以将上述 SQL 中的 in 操作转化为 join 操作。

上述 SQL 经过子查询提升可得优化后的关系代数表达式：

$$ \pi(J \bowtie_{j.city=s.city} \pi_{S.City}(S)) $$

五、物理优化之单表扫描

通过基于关系代数等价运算规则进行逻辑优化，关系数据库可以得到一个优化后的关系代数表达式，一定程度决定数据库该以什么顺序执行什么样的操作，将 SQL 查询语言从“做什么（what）”向“怎么做（how）”迈出了一大步。数据库从关系代数表达式中虽然可以知道该以什么顺序做选择、投影，但是其依然不清楚具体的每一步该怎么执行，比如对单个表是采用顺序遍历还是索引遍历？两表连接是采用 hash 连接还是嵌套循环连接？以及多张表的连接顺序该如何确定？

因此数据库系统需要对关系代数表达式做进一步处理，利用页面 I/O 花费、元组 CPU 执行时间、表的物理信息（页面数量、元组数量、属性上是否有索引）等信息，对不同的物理执行方式做代价评估，最终选出一条代价最小的物理执行路径，这条路径不仅包含操作的顺序，也指明了实现操作的方式。

由于单表扫描是复合操作的基础（可以理解为查询执行树的叶节点），是一个查询执行计划的基础组成成分，因此我们先对单表扫描操作进行学习研究。

首先利用 explain 命令执行 SQL：

```
testdb=# explain select Jno, Jname from J;
                        QUERY PLAN
-----------------------------------------------------------
```

```
 Seq Scan on j  (cost=0.00..78.00 rows=5000 width=11)
(1 行记录)
```

在 explain 命令作用下 `select Jno, Jname from J` 命令并未真正执行，而是输出查询执行计划（QUERY PLAN）。现在对输出的执行计划每个数据项进行逐一介绍：

- **Seq Scan on j**: 表示对j表进行顺序扫描，即从表的第一个页的第一个元组扫描到最后一个元组。由于没有选择条件，数据库查询优化器对 $ \pi_{Pno, Pname}(J) $ 选择了顺序扫描的物理执行方式。
- **cost=0.00..78.00**: 表示该条执行计划的代价估算，第一个'0.00'表示执行计划的启动成本；'78.00'表示从启动到完成的总代价估计，一般有三部分代价构成：启动代价 + I/O代价 + CPU时间代价。
- **rows=5000 width=11**：表示此计划节点输出的估计总行数，以及输出元组的估计平均宽度。

执行下述命令，可以看到去掉投影操作后，虽然代价估计值与输出行数没有变化，但是输出的元组宽度却增加到16。如果对J表的扫描操作是其他复合操作（例如上文关系代数中的连接操作）的基础操作，会导致中间结果膨胀，可能会造成更多内存消耗，增加 I/O。

```
testdb=# explain select * from J;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on j  (cost=0.00..78.00 rows=5000 width=16)
(1 行记录)
```

对查询添加限定条件 "J.Jno > 4950"，检索 Jno 大于4950的记录。由于 J.Jno 列存在 B+ 树索引，且元组按照 Jno 大小顺序存放，因此查询优化引擎选择 Index Scan（索引扫描的方式）检索数据。索引扫描的原理是，利用索引根据查找键快速定位到元组的位置，可以消减不必要的元组扫描，但同时会带来索引扫描与随机 I/O 的额外开销。根据 QUERY PLAN 中代价估算值 cost 可知，根据选择条件并使用索引时执行 $\sigma_{J.Jno > 4950}(J)$ 的估计 cost 为9.16，相较表的顺序扫描（78.00 + 每个元组执行操作 J.Jno > 4950 的代价）大大降低了代价。

```
testdb=# explain select * from J where J.Jno > 4950;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using j_pkey on j  (cost=0.28..9.16 rows=50 width=16)
   Index Cond: (jno > 4950)
(2 行记录)
```

经过上述三个例子可以大致得到结论：完成单表扫描获取符合条件的元组实际存在多种执行方式，具体选择何种物理执行方式，PostgreSQL 查询优化器会根据能否使用索引，I/O 与 CPU 花费代价 cost 等标准对不同的物理执行方式进行代价评估，选择代价最小的物理执行方式，作为 SQL 查询的物理执行代价。

接下来我们简要介绍 PostgreSQL 代价估算的方法。

PostgreSQL 查询优化代价估算基于 CPU 开销和 I/O 开销，其计算公式如下：

$$\text{总代价} = \text{启动代价} + \text{IO代价} + \text{CPU代价}$$

为了提供较为准确的代价评估，PostgreSQL 在系统中内置了多种代价常数，其中基础的5个参数如下所示：

- **seq_page_cost = 1**：表示从磁盘顺序读入页面的代价

- **random_page_cost = 4**：表示从磁盘随机读入页面的代价
- **cpu_tuple_cost = 0.01**: cpu处理每个元组的代价
- **cpu_index_tuple_cost = 0.005**: 在索引扫描期间CPU处理每个索引条目的代价
- **cpu_operator_cost = 0.0025**: CPU在查询处理期间执行每个操作符或函数的代价

例如上述第1条与第2条 SQL 语句，查询优化器选择对表J进行顺序扫描（从磁盘中顺序读入页面），之后 CPU 读取每一条元组，且没有任何判断条件。因此，对表J的顺序扫描代价计算公式为：

$$ Cost(seq\_scan(J)) = relpages * seq\_page\_cost + reltuples * cpu\_tuple\_cost $$

我们通过运行下述命 SQL 命令获取表 J 的页面和元组数量(注意 "j" 要小写)：

```
testdb=# SELECT relpages, reltuples FROM pg_class WHERE relname = 'j';
 relpages | reltuples
----------+-----------
       28 |      5000
```

带入代价计算公式可得： 28 * 1 + 5000 * 0.01 = 78.00，和输出的查询计划代价计算结果相同。

输入下述命令，当判断条件改为 J.Jno > 0 时，PostgreSQL 优化器选择对表 J 进行顺序扫描（seq scan）从第一条元组顺序遍历到最后一条元组。下述语句相较第2条语句都采用顺序扫描，不同之处在于对每条元组多出来一次判断处理（J.Jno > 0）。因此，代价计算公式为：

$$ Cost(seq\_scan(J)) = relpages * seq\_page\_cost + reltuples * cpu\_tuple\_cost + reltuples * cpu\_operator\_cost $$

带入代价计算公式可得：28 * 1 + 5000 * 0.01 + 5000 * 0.0025 = 90.50，和输出的查询计划代价计算结果相同。

```
testdb=# explain select * from J where J.Jno > 0;
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on j  (cost=0.00..90.50 rows=5000 width=16)
   Filter: (jno > 0)
(2 行记录)
```

针对第3条 SQL 语句中的索引扫描，其查询计划代价评估更为复杂一些。首先是不同的索引类型，有着不同的代价评估函数；其次，数据的分布也会影响到索引扫描的效率（顺序分布，随机分布、正态分布等）。
不过不管是哪种情况，索引扫描也要遵循“总代价 = IO 代价 + CPU 代价”的基本原理。本次 Lab 针对索引扫描代价评估仅讨论上述5个参数构成的代价模型，不讨论更为复杂的针对索引性能的估计，仅关注索引扫描和顺序扫描代价模型主要的异同。

索引扫描的代价模型为：

$$ Cost(Index Scan(j)) = x1 * seq\_page\_cost + x2 * cpu\_tuple\_cost + x3 * cpu\_operator\_cost + x4 * random\_page\_cost + x5 * cpu\_index\_tuple\_cost $$

由于再搜索符合条件（j.jno > 4950）时使用了索引，因此并不需要像顺序扫描遍历全部的元组，可以利用索引定位一个范围，大大消减遍历的元组的数量，因此 x1 x2 x3 三个系数相较顺序扫描减少。但是索引扫描又会带来新的代价，对索引进行搜索时，需要访问索引页面内的索引记录，因此带来 $ x5 * cpu_index_tuple_cost $ 的开销；由于 PostgreSQL 中索引页面与数据页面不在同一个数据文件中，因此当从索引叶节点访问对应的数据页面时，需要一次随机页面访问（随机I/O），对于磁盘来说，随机 I/O 的成本要远大于顺序 I/O，因此带来 $ x4 * random_page_cost $ 的额外开销。

第3条 SQL 语句的代价估计为9.16（阿里云数据库上的代价估计为3.36，可能收到了硬件环境的影响），由于索引扫描代价评估涉及到的参数过多以及关键信息的缺失，暂时无法判断代价模型中各参数的具体数值。个人猜测，至少包含一次随机页面访问的代价。

通过上述4个示例展示经过物理优化的单表扫描查询计划，我们可以有一个初步的认识：物理优化是在逻辑优化的基础上，通过获取系统数据（表页面的数量、元组的数量、是否有索引、查询条件等），选择合适的物理执行方式。通过对前三条 SQL 查询执行计划、执行评估进行分析，第3条 SQL 相较第2条 SQL 多了选择条件，减少了返回的元组数量（预测返回的rows从5000减为50）；第1条 SQL 相较第2条 SQL 多了映射条件，减少了返回元组的大小（预测的 width 由16减为11），证明了逻辑优化阶段选择与投影下推可以有效地降低生成中间结果的时间、空间复杂度，提升 SQL 效率。

六、物理优化之两表连接

关系代数中重要的操作有选择、投影与连接三部分。上一节讲了选择、投影对单表扫描物理执行计划的影响，本节讲解两表连接运算的物理执行方式。

PostgreSQL 中包含三种两表连接运算方式，分别是：嵌套循环连接（nested loop join）；排序归并连接（mergejoin）；hash 连接（hash join）。其中嵌套循环连接与 hash 连接操作可以观看水杉课程第六章6.6-6.8节的视频，下面简要讲解排序归并连接。

归并排序连接简称为归并连接，其算法执行流程和归并排序有相似之处。归并连接在做两表连接操作之前，要求两个表分别根据连接属性进行排序，排序后采用类似归并排序中对两个有序子数组合并的过程，对两个有序表进行连接操作。具体流程如下：

- 首先以目标 SQL 中指定的谓词条件（如果有的话）去访问表 T1，然后对访问结果按照表 T1 中的连接列来排序，排好序后的临时表我们记为 S1。
- 接着以目标 SQL 中指定的谓词条件（如果有的话）去访问表 T2，然后对访问结果按照表 T2 中的连接列来排序，排好序后的临时表我们记为 S2。
- 最后对结果集1和结果集2执行合并操作：首先有两个指针 P1、P2 分别指向两个中最小的元组，如果相等则做连接操作，且将其中一个指针向后移动一个元组；如果不相等，例如 P1 < P2，则将较小的元组的指针 P1 向后移动一个元组，之后继续做比较。重复上述操作，直到其中一张表遍历全部的元组，结束连接操作。

在归并连接的过程中，如果连接的列上存在索引，则可利用索引对连接列进行排序，可以降低排序造成的时间花费。

下面展示几个 SQL 语句使用不同连接算法的例子并解释他们实际的执行流程：

注：spj.qty 属性均匀分布在 [1, 500] 整数域中。

输入下述命令：

```
testdb=# explain select P.Pno, P.Pname  from P, SPJ  where SPJ.Pno =
P.Pno and SPJ.qty > 300;
                            QUERY PLAN
------------------------------------------------------------------
 Hash Join  (cost=2.13..2004.06 rows=40365 width=11)
   Hash Cond: (spj.pno = p.pno)
   ->  Seq Scan on spj  (cost=0.00..1887.00 rows=40365 width=4)
         Filter: (qty > 300)
   ->  Hash  (cost=1.50..1.50 rows=50 width=11)
         ->  Seq Scan on p  (cost=0.00..1.50 rows=50 width=11)
(6 行记录)
```

根据 explain 命令输出的 QUERY PLAN 可知，上述 SQL 命令执行流程如下：

1. 首先对p表进行顺序扫描，并根据Pno值进行hash，构造出hash桶。（该步骤由于没有选择条件，索引预估输出的元组数量为p表元组总数量50，并且由于符合投影串接率，将pno pname投影下推到此步骤，因此生成的中间元组大小只有11，减少了中间结果大小）

```
->  Hash  (cost=1.50..1.50 rows=50 width=11)
      ->  Seq Scan on p  (cost=0.00..1.50 rows=50 width=11)
```

2. 接下来对spj表进行顺序扫描，并筛选出 qty > 300的元组。（预估符合条件的元组数量是40365条，宽度为4，但是SPJ表的总宽度远大于4，说明SQL经过优化，此步骤做了投影操作只使用了spj.pno列数据，其余列均未包含在中间结果中，大大降低了中间结果大小）

```
->  Seq Scan on spj  (cost=0.00..1887.00 rows=40365 width=4)
      Filter: (qty > 300)
```

3. 最后将第2步得到的符合条件的元组，利用hash join操作，对每个元组进行hash，根据hash值与第1步对应的hash桶内的元组进行匹配，匹配条件为(spj.pno = p.pno)。（由于pno是p表的主键且唯一，因此对于SPJ表中任一元组的pno号，在p表中有且仅有一条元组匹配。因此预估的输出结果为40365条元组，等于符合筛选条件的SPJ表元组数量）

```
Hash Join  (cost=2.13..2004.06 rows=40365 width=11)
  Hash Cond: (spj.pno = p.pno)
```

再测试另一个 SQL 查询：

```
testdb=# explain  select P.Pno, P.Pname  from P, SPJ  where SPJ.Pno =
P.Pno and SPJ.qty > 499;
                                QUERY PLAN
------------------------------------------------------------------------
------
 Nested Loop  (cost=0.15..1902.56 rows=188 width=11)
   ->  Seq Scan on spj  (cost=0.00..1887.00 rows=188 width=4)
         Filter: (qty > 499)
   ->  Memoize  (cost=0.15..0.23 rows=1 width=11)
```

```
         Cache Key: spj.pno
         Cache Mode: logical
         ->  Index Scan using p_pkey on p  (cost=0.14..0.22 rows=1
width=11)
               Index Cond: (pno = spj.pno)
```

根据 explain 命令输出的 QUERY PLAN 可知，上述 SQL 命令执行方式如下：

1. 最外层是嵌套循环连接（Nested Loop），表明 SPJ 与 P 表使用的连接算法是嵌套循环连接

```
Nested Loop  (cost=0.15..1902.56 rows=188 width=11)
```

2. 嵌套循环中的外层循环是对 SPJ 的单表扫描，采用的是顺序扫描，扫描的同时应用选择条件（spj.qty > 499）。（由于 spj 表在 qty 属性上无索引，因此选择顺序扫描，且将选择条件下推到顺序扫描的过程中，可消减参与连接操作的元组数量。预估符合选择条件的元组数量为188）

```
->  Seq Scan on spj  (cost=0.00..1887.00 rows=188 width=4)
      Filter: (qty > 499)
```

3. 将外层循环中得到的参与连接的元组，与内层表中符合条件的元组进行连接。根据最下方的执行计划可知，根据 spj 中符合条件的元组的 spj.pno，利用索引对 p 表进行索引扫描，快速的获取 p.pno = spj.pno 的元组，之后将二者进行连接操作输出结果。除了对表 p 的索引扫描，执行计划中还有 Memoize 物理操作，这个操作是用于嵌套循环中对内层表访问的优化。其作用是，对映射关系 y = f(x)，Memoize 缓存 x y 的对应关系，因此对到来的 x，不用再通过 f(x) 计算得到 y，而是直接返回 y。在本示例中，x 即为 pno 值，f(x) 即索引扫描得到符合（pno = spj.pno）条件的元组这一过程，y 即为 p 中符合条件的元组。如果没有 memoize 操作，每次对 p 表找到符合条件的元组都需要进行索引扫描，需要花费额外的时间；如果利用 memoize 操作，可以将 pno 与 p 元组的映射关系缓存下来，对某 pno 值除了第一次做索引扫描外，后续相同的 pno 值均可通过缓存的映射关系直接返回对应的元组，可减少索引扫描的开销。

```
->  Memoize  (cost=0.15..0.23 rows=1 width=11)
      Cache Key: spj.pno
      Cache Mode: logical
      ->  Index Scan using p_pkey on p  (cost=0.14..0.22 rows=1
width=11)
            Index Cond: (pno = spj.pno)
```

接下来介绍两表连接操作的代价估计模型。

由于两表连接算法的实际代价估算模型非常复杂，会受到算法具体实现方式、硬件环境、数据的分布（顺序、随机或者符合某个概率模型）、数据选择率等因素的影响。因此下面从算法执行流程分析，建立一个抽象的代价估算模型，忽略掉具体的代价细节，下述代价模型均假设S为左连接表，P 为右连接表。

1. 嵌套循环连接

Cost(nested loop join(S, P)) = Scan(S) + Scan(P) * Select(S)

2. 归并连接

Cost(merge join(S, P)) = Sort(S) + Sort(P) + Scan(S_tmp) + Scan(P_tmp)

3. Hash连接

Cost(hash join(S, P)) = Createhash(P) + Scan(S) + Select(S) * Number(P) * 选择率

Scan 表示对表进行扫描操作的代价。不同的物理执行方式有不同的代价，例如嵌套循环连接中，Scan(P) 可以采用索引扫描的方式进行优化，并且可以采用 Memoize 缓存映射关系进行优化。但是对于归并连接，Scan 是对 S P 表排序后的临时表进行扫描，由于有序且无索引，因此采用顺序扫描的方式。

Select 表示某张表实际参与连接操作的元组数量。

Sort 表示对表进行排序操作的代价。如果排序的列上存在有序索引，则可利用索引进行排序，可降低 Sort 的代价。

Createhash 表示根据表中某个属性创建 hash 桶的代价。一般来说，P 表的元组数量不变且属性分布不变，其创建 hash 桶的代价也不变。

Scan(S) * Number(P) * 选择率 表示将表 S 与构造 Hash 桶后的表 P 进行 hash 连接的代价。选择率表示对于表 P 某个属性的某个取值，对应元组占表 P 所有元组的比例。 如果对表 S 存在选择、投影等操作且选择列具有索引，可以在 Scan(S) 阶段采用索引扫描以减少代价。

七、练习

习题1：

请在保证语义等价的前提下，利用谓词下推、子查询提升对下述 SQL 语句进行逻辑优化，写出优化后的关系代数表达式。

```
select
    J.Jname, J.city
from
    J, SPJ
where
        J.Jno = SPJ.Jno
    and SPJ.qty > 400
    and J.city in
            (select city
             from S
             where S.Sname = '东方');
```

解答：

1.子查询提升： 将子查询结果计算一次并且缓存为临时表FilteredCities，独立计算以便减少重复计算；利用索引加速计算进程。

2.谓词下推： 用谓词下推将SPJ.qty>400的过滤条件提前到扫描阶段，减少参与连接的数据量。

In [ ]:
```
#优化后代码
select
    j.jname, j.city
from
    j,
    (select jno
     from spj
     where qty > 400) as filtered_spj
where
    j.jno = filtered_spj.jno
    and j.city in
        (select city
         from s
         where s.sname = '东方');
```

习题2：

请分别运行以下两条命令：

```
explain select * from SPJ full outer join P on SPJ.Pno = P.Pno and
SPJ.QTY > 400 and P.weight > 500;
explain select * from SPJ full outer join P on SPJ.Pno = P.Pno where
SPJ.QTY > 400 and P.weight > 500;
```

请根据输出的查询计划，解释当存在对连接表的选择谓词时，where 和 on操作的区别：

解答：

1.on在连接阶段应用，对连接的匹配行起作用；where在连接之后应用，对最终结果起作用。

2.on适用于需要精确控制连接逻辑的场景，并保留未匹配行；但where适用于需要在最终结果集上进一步过滤的场景，且可能改变full outer join的语义。

3.on更高效，减少连接时处理的行数；where较低效，因连接所有行后再进行过滤，处理的行数更多。

习题3;

在“物理优化之单表扫描”一节中，第3条和第4条查询语句仅改变了判断条件（J.Jno > 4950; J.Jno > 0），SQL 优化器却选择了不同的单表扫描方式。 可以尝试下述操作寻找规律：

```
explain select * from J where J.Jno > n;
```

n 由5000逐渐降低（在本地 PostgreSQL14 运行时在 n=3210 附近；在阿里云数据库环境运行时在 n=1678 附近），对表 J 的扫描操作由索引扫描变为了顺序扫描。

根据第5节中对顺序扫描、索引扫描执行过程以及代价估计的分析，请简要分析出现上述现象的原因。（不需要具体的数值计算）

解答：

1.数据的分布选择： n较大时，条件J.Jno>n会匹配更少的行，选择性较低，这个时候索引扫描会更高效。索引扫描会逐行读取目标行，但如果需要访问的行数占总行数的比例较大，逐行读取的成本会变得更高。

2.表大小和页缓存： 表J较大的时候，顺序扫描可以充分利用预读机制，批量加载数据，减少I/O开销。当n降低到一定值时，匹配的行分布在更大范围的页内，顺序扫描的批量读取优势逐渐显现，索引扫描反而因为频繁跳页而变得不经济。

3.索引维护和代价估计： 索引扫描不仅需要扫描索引结构，还要通过索引来定位数据，这个过程会产生额外的开销，尤其是如果需要访问大量的行时。优化器会估算使用索引的代价，并考虑是否需要频繁地回表。回表的次数太多，索引扫描会变得低效。当n较小时，匹配的行数增多，回表操作变得频繁，索引扫描的代价会超出顺序扫描的代价，此时优化器会选择顺序扫描。

习题4：

在“物理操作之两表连接”一节中，测试的两条 SQL 语句仅改变了选择条件（qty > n），但是却选择了不同的 join 算法。

当 n 不断增加时，spj 表符合条件的元组越来越少，则参与连接的元组也相应减少，右连接表 P 参与连接的元组不变。

尝试根据嵌套循环连接与 Hash 连接执行过程与代价估计模型，简要分析当 n 增加时，hash join 算法和 nested loop 连接算法的代价估计模型变化情况以及出现上述现象的原因。（不需要具体的数值计算）

解答：

1.hash join 算法： 总扫描次数O(|R|+|S|)。当n不断增加时，spj表符合条件的元组减少，哈希连接在构建哈希表时只处理符合条件的元组，因此哈希表的大小随之减少，扫描右表的次数保持不变。由于哈希连接只需要对符合条件的元组构建哈希表，这使得其代价随着n的增加而降低，尤其是当SPJ表中的符合条件的元组较少时，哈希连接更具优势。

2.nested loop 连接算法： 总扫描次数O(|R|*|S|)。当n增加时，SPJ表符合条件的元组数量减少，外层循环扫描次数减少，代价相应减少。但是由于内层循环对右表进行全表扫描，右表的扫描次数不受影响。因此，如果右表P较大，嵌套循环连接的代价仍然可能较高。当n增大时，SPJ表的有效元组减少，这使得外层扫描次数降低，从而减少了代价。

3.总结： （1）嵌套循环连接：随着n增加，左表符合条件的元组减少，外层扫描次数减少，代价降低。但对于较大的右表，内层扫描的代价可能仍然较高。（2）哈希连接：随着n增加，左表符合条件的元组减少，哈希表的大小减少，代价降低。哈希连接对大表尤其有效，即使左表符合条件的元组减少，代价仍然较低。

参考资料

1. PostgreSQL官方文档： https://www.postgresql.org/docs/14/using-explain.html
2. PostgreSQL优化器入门：http://t.zoukankan.com/traditional-p-12826605.html
3. PostgreSQL 数据库内核分析，彭智勇等

4. PostgreSQL 技术内幕：查询优化深度探索，张树杰
5. 数据库查询优化器的艺术：原理解析与 SQL 性能优化，李海翔等