实习4

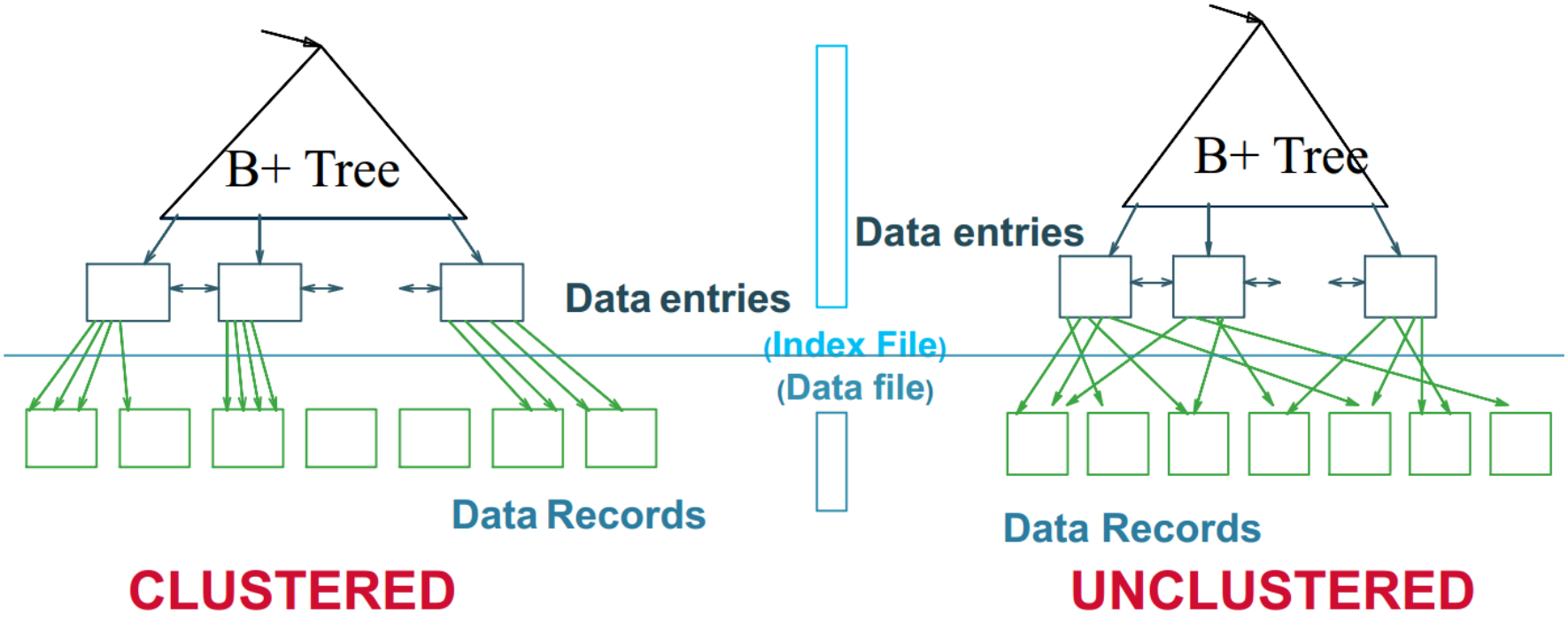
空间查询处理和优化

代价估计与索引

- 健身俱乐部数据库
  - Gym (gid, name, city)
  - Member (mid, name, is_student, birthdate, city)
  - Visits (timestamp, mid, gid)
- Member和Visits在数据库中的统计信息如下
  - T(Member) = 500, B(Member) = 100, V(Member, city) = 10, V(Member, is_student) = 2
  - T(Visits) = 5000, B(Visits) = 400, V(Visits, mid) = 500
- $\sigma_{a=?}(R)$
  - 查询结果的数量为 T(R) / V(R, a)
  - 比如每个城市的会员平均为 500 / 10 = 50人

代价估计与索引

- 聚集和非聚集索引



Every table can have **only one** clustered and **many** unclustered indexes

代价估计与索引

- 估计查询的cost (最差情况下数据块的读取数量)

  Select name From Member

  Where city = '杭州' and is_student = True

- 没有索引
  - 不知道数据如何在硬盘中存储，需要顺序扫描整个数据库
  - 最差情况下，cost = B(Member)
- Member的city属性上有非聚集索引
  - 利用city属性的非聚集索引，每个杭州会员可能在不同的数据块上
  - 最差情况下，cost = T(Member) / V(Member, city)

代价估计与索引

- 估计查询的cost (最差情况下数据块的读取数量)

  Select name From Member

  Where city = '杭州' and is_student = True

- Member的city属性上有非聚集索引
  - 利用city属性的非聚集索引，每个杭州会员可能在不同的数据块上
  - 最差情况下，cost = T(Member) / V(Member, city)
- Member的city属性上有聚集索引
  - 利用city属性的聚集索引，杭州会员在磁盘上存储在连续的数据块
  - 最差情况下，cost = B(Member) / V(Member, city)

代价估计与索引

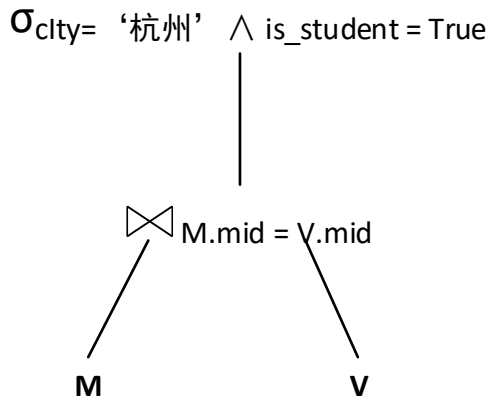
- 估计查询的cost (最差情况下数据块的读取数量)

  Select name From Member

  Where city = ‘杭州’ and is_student = True

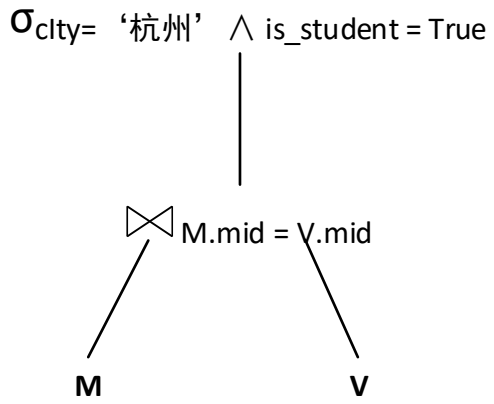
- Member的is_student属性上有非聚集索引
  - 利用is_student属性的非聚集索引，每个学生会员可能在不同的数据块上
  - 最差情况下，cost = T(Member) / V(Member, is_student)
- Member的(city, is_student)属性上有非聚集索引
  - 最差情况下，cost = T(Member) / V(Member, city) / V(Member, is_student)

代价估计与索引



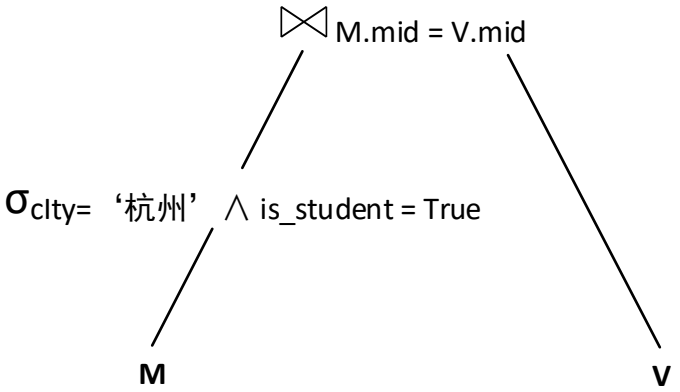
- 没有索引，使用nested loop的最小I/O cost
  - B(Member) < B(Visits)
  - B(Member) + B(Member) * B(Visits)
- Visits的mid属性上有非聚集索引，使用nested loop with index的最小I/O cost
  - B(Member) + B(Member) * T(Visits) / V(Visits, mid)

代价估计与索引



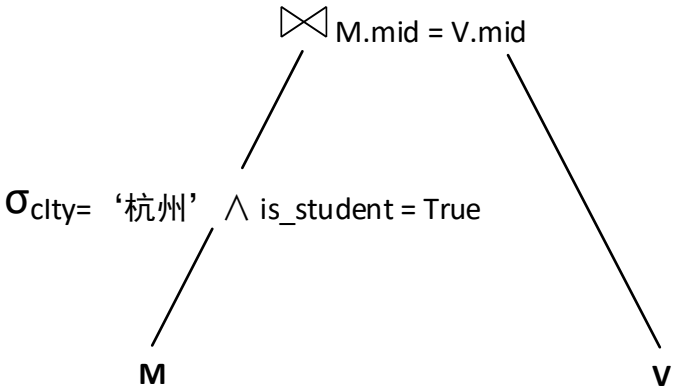
- Visits的mid属性上有非聚集索引，使用nested loop with index的最小I/O cost
  - B(Member) + B(Member) * T(Visits) / V(Visits, mid)
- Visits的mid属性上有聚集索引，使用nested loop with index的最小I/O cost
  - B(Member) + B(Member) * B(Visits) / V(Visits, mid)
  - 数据块是整个读取，所以0.8个数据块，就是读1个数据块

代价估计与索引



- Member的(city,is_student)属性上有非聚集索引，使用nested loop的的I/O cost
  - cost1 = T(M) / V(M, city) / V(M, is_student) -- *3.1.5*
  - cost = cost1 + cost1 * B(Visits)

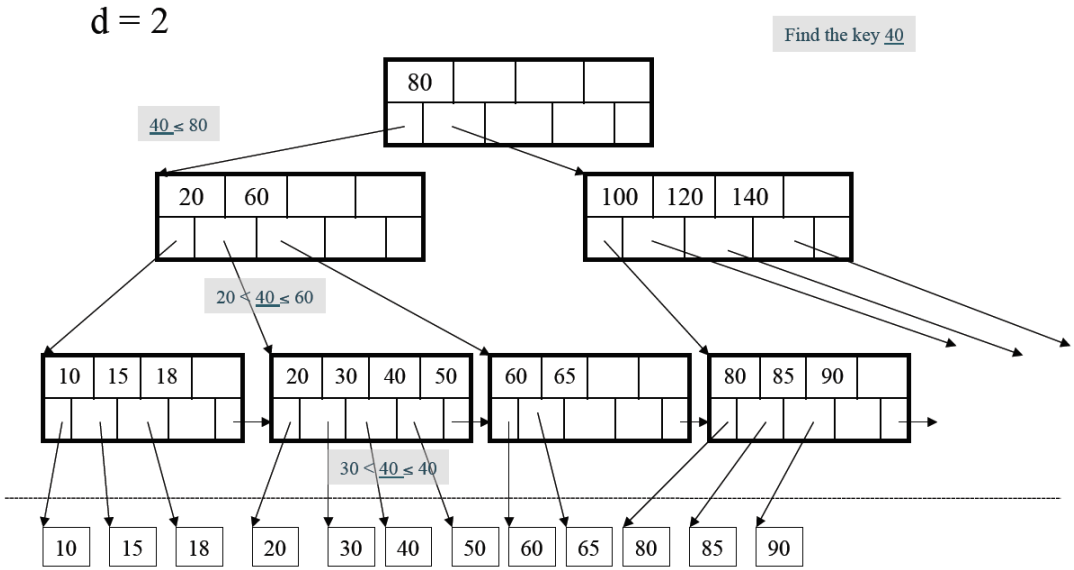
代价估计与索引



- Member的(city,is_student)属性上有非聚集索引，Visits的mid属性上有非聚集索引，使用nested loop的的I/O cost
  - cost1 = T(M) / V(M, city) / V(M, is_student) = 25
  - cost2 = T(V) / V(V, mid) = 10
  - cost = cost2 + cost2 * cost1 = 10 + 10 * 25 = 260

代价估计与索引

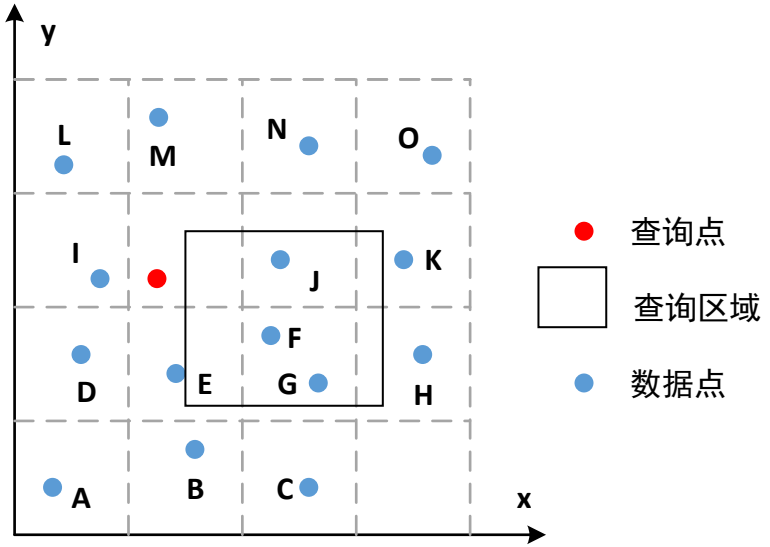
- 为什么mid聚集索引会使用户签到变慢，即插入一个新的Visit变慢？
  - 聚集索引要求磁盘数据存储按照mid顺序，当用户签到时，该签到记录需要插入到mid所在磁盘页，可能会导致数据在磁盘上移动
  - 类似于10插入排序数组4, 5, 9, 12, 15
  - 通过B+树可以快速查找mid插入位置，即所在的磁盘页



代价估计与索引

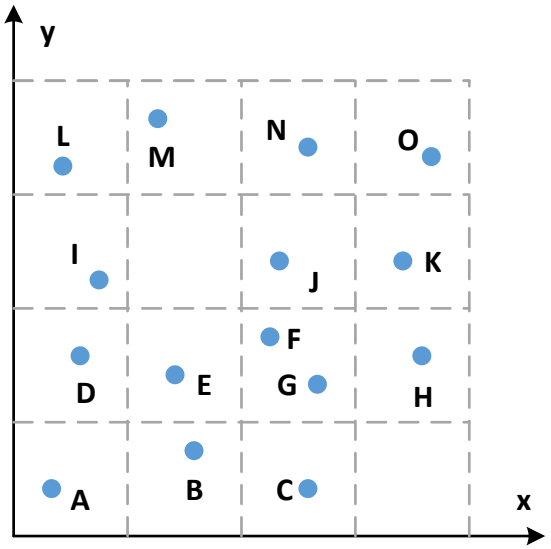
- 你构建了索引选择工具来度量某个索引对常用查询的性能提升(越高越好)。索引1(city, is_student)带来的性能提升是10，索引2(city, is_student, birthday)带来的性能提升是12，索引3(city, birthday)带来的性能提升是7。假设你只能保留2个索引，你会选择哪两个索引，理由是什么？
  - 首先选择索引2，因为对常用查询的性能提升最高
  - 其次选择索引3，因为虽然索引1带来的性能提升比索引3高，但所有使用索引1的加速，都能通过索引2来加速，所以当选择索引2后，索引1是冗余的

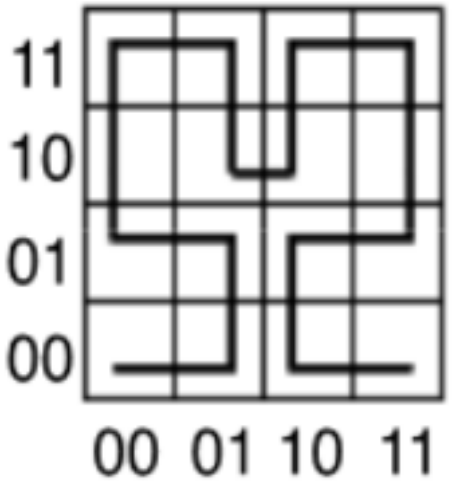
空间填充曲线



- Heap file存储，乱序存储，15个几何对象，每个数据块最多2个几何对象
  - Point Query: 8
  - Range Query: 8
  - Nearest Neighbor Query: 8

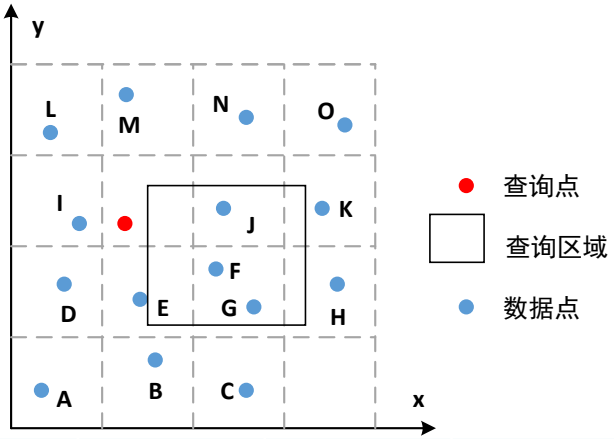
空间填充曲线





- 构建4x4的Hilbert Curve，数据点按照Hilbert value的顺序存储，使用()表示一个数据块，给出几何点在数据库中的存储顺序
  - (A, B) (E, D) (I, L) (M, J) (N, O) (K, H) (F, G) (C)

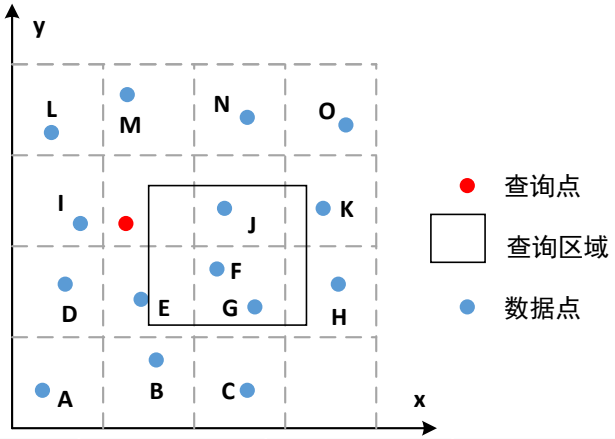
空间填充曲线



数据块	0	1	2	3	4	5	6	7
几何点	(A, B)	(E, D)	(I, L)	(M, J)	(N, O)	(K, H)	(F, G)	(C)
H值	0, 1	2, 3	4, 5	6, 8	9, 10	11,12	13,13	14

- Point Query (二分查找是在数据块0-7中查询)
  - 查询点的H = 7
  - Block ceil((0 + 7)/2) = 4 → H = (9, 10)
  - Block ceil((0 + 4)/2) = 2 → H = (4, 5)
  - Block ceil((2 + 4)/2) = 3 → H = (6, 8)

空间填充曲线



数据块	0	1	2	3	4	5	6	7
几何点	(A, B)	(E, D)	(I, L)	(M, J)	(N, O)	(K, H)	(F, G)	(C)
H值	0, 1	2, 3	4, 5	6, 8	9, 10	11,12	13,13	14

- Range Query
  - 查询区域的H = 2, 7-8, 11-13

空间填充曲线

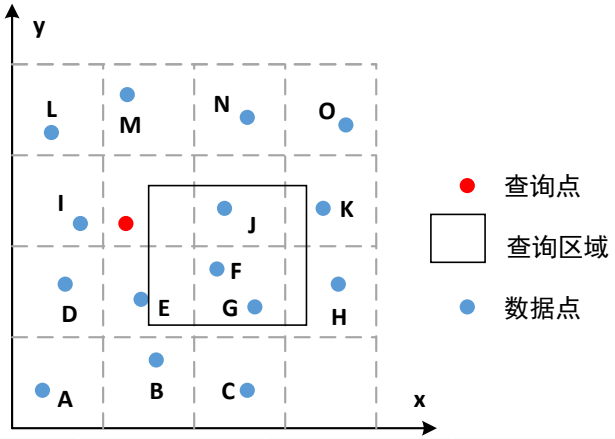
数据块	0	1	2	3	4	5	6	7
几何点	(A, B)	(E, D)	(I, L)	(M, J)	(N, O)	(K, H)	(F, G)	(C)
H值	0, 1	2, 3	4, 5	6, 8	9, 10	11,12	13,13	14

- Range Query H = 2, 7-8, 11-13
  - H = 2
    - Block (0+7)/2 = 4 → H = (9, 10)
    - Block (0+4)/2 = 2 → H = (4, 5)
    - Block (0+2)/2 = 1 → H = (2, 3)
  - H = 7
    - Block (0+7)/2 = 4 → H = (9, 10)
    - Block (0+4)/2 = 2 → H = (4, 5)
    - Block (2+4)/2 = 3 → H = (6, 8)
  - H = 11
    - Block (0+7)/2 = 4 → H = (9, 10)
    - Block (4+7)/2 = 6 → H = (13, 13)
    - Block (4+6)/2 = 5 → H = (11, 12)

内存足够大，缓存Block
整个过程需要读取
Block 1, 2, 3, 4, 5, 6
Cost = 6

内存不够大，不缓存Block
整个过程需要读取
Block 4 2 1 4 2 3 4 6 5
Cost = 9

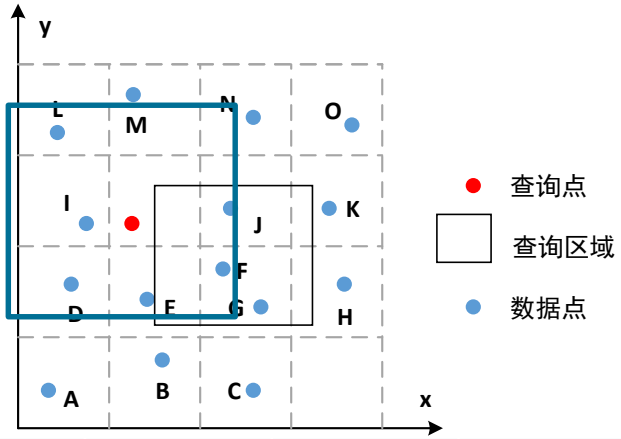
空间填充曲线



数据块	0	1	2	3	4	5	6	7
几何点	(A, B)	(E, D)	(I, L)	(M, J)	(N, O)	(K, H)	(F, G)	(C)
H值	0, 1	2, 3	4, 5	6, 8	9, 10	11,12	13,13	14

- Nearest Neighbor Query
  - 查询点的H = 7
  - Block ceil((0 + 7)/2) = 4 → H = (9, 10)
  - Block ceil((0 + 4)/2) = 2 → H = (4, 5)
  - Block ceil((2 + 4)/2) = 3 → H = (6, 8) → 计算与M,J的距离

空间填充曲线



数据块	0	1	2	3	4	5	6	7
几何点	(A, B)	(E, D)	(I, L)	(M, J)	(N, O)	(K, H)	(F, G)	(C)
H值	0, 1	2, 3	4, 5	6, 8	9, 10	11,12	13,13	14

- Nearest Neighbor Query
  - 查询点的H = 7
  - 计算与M,J的距离
  - 查询区域的H = 2-9, 13

空间填充曲线

数据块	0	1	2	3	4	5	6	7
几何点	(A, B)	(E, D)	(I, L)	(M, J)	(N, O)	(K, H)	(F, G)	(C)
H值	0, 1	2, 3	4, 5	6, 8	9, 10	11,12	13,13	14

- Point Query H = 7
  - Block (0 + 7)/2 = 4 → H = (9, 10)
  - Block (0 + 4)/2 = 2 → H = (4, 5)
  - Block (2 + 4)/2 = 3 → H = (6, 8)
- Nearest Neighbor Query H = 2-9, 13
  - H = 2
    - Block (0+7)/2 = 4 → H = (9, 10)
    - Block (0+4)/2 = 2 → H = (4, 5)
    - Block (0+2)/2 = 1 → H = (2, 3)
  - H = 13
    - Block (0+7)/2 = 4 → H = (9, 10)
    - Block (4+7)/2 = 6 → H = (13, 13)

内存足够大，缓存Block
整个过程需要读取
Block 1, 2, 3, 4, 6
Cost = 5

关系代数优化

- 在关系数据库系统中，cost主要是I/O cost，即数据读取次数（注意和空间数据库系统的差异）。在计算cost时，做了以下假设：1. 存储系统没有cache数据，无论是buffer management还是磁盘上的cache；2. 自然连接实现方式，是基于什么算法？通过构造等价的关系代数表达式，优化以下查询。
  - 先选择，后连接 (减少连接时的行数)
  - 多个关系连接时，先数据量较小的两个关系的连接

关系代数优化

- 先选择，后连接
- 多个关系连接时，先数据量较小的两个关系的连接

$\sigma_{c=0}(\Pi_{a,c}(\sigma_{b=0}((\ (\ R(a,b)\ ) \bowtie_b (\ S(b,c)\ )\ ) \bowtie_c (\ T(c,d)\ ))))$

[(0, 0), (6, 0), (2, 0), (8, 0), (4, 0)]

Total Reads: 4365

- $\sigma_{c=0}$ [tuples read in: 15 out: 5]
  - $\Pi_{['a','c']}$ [tuples read in: 75 out: 15]
    - $\sigma_{b=0}$ [tuples read in: 375 out: 75]
      - $\bowtie_c$ [tuples read in: 3250 out: 375]
        - $\bowtie_b$ [tuples read in: 650 out: 125]
          - R(a,b) has 25 tuples
          - S(b,c) has 25 tuples
        - T(c,d) has 25 tuples

$\Pi_{a,c}((\ \sigma_{c=0}((\ \sigma_{b=0}(R(a,b))\ ) \bowtie_b (\ \sigma_{b=0}(S(b,c))\ ))\ ) \bowtie_c (\ \Pi_c(\sigma_{c=0}(T(c,d)))\ ))$

[(8, 0), (2, 0), (0, 0), (6, 0), (4, 0)]
True

Total Reads: 370

- $\Pi_{['a','c']}$ [tuples read in: 5 out: 5]
  - $\bowtie_c$ [tuples read in: 10 out: 5]
    - $\sigma_{c=0}$ [tuples read in: 25 out: 5]
      - $\bowtie_b$ [tuples read in: 30 out: 25]
        - $\sigma_{b=0}$ [tuples read in: 25 out: 5]
          - R(a,b) has 25 tuples
        - $\sigma_{b=0}$ [tuples read in: 125 out: 25]
          - S(b,c) has 25 tuples
    - $\Pi_{['c']}$ [tuples read in: 25 out: 5]
      - $\sigma_{c=0}$ [tuples read in: 125 out: 25]
        - T(c,d) has 25 tuples

关系代数优化

- 先选择，后连接
- 多个关系连接时，先数据量较小的两个关系的连接

$\Pi_{a,c}((\sigma_{c=0}((\sigma_{b=0}(\text{R(a,b)})) \bowtie_b (\sigma_{b=0}(\text{S(b,c)}))) \bowtie_c (\Pi_c(\sigma_{c=0}(\text{T(c,d)}))))$

```
[(8, 0), (2, 0), (0, 0), (6, 0), (4, 0)]
True
```

Total Reads: 370

- $\Pi_{['a','c']}$ [tuples read in: 5 out: 5]
  - $\bowtie_c$ [tuples read in: 10 out: 5]
    - $\sigma_{c=0}$ [tuples read in: 25 out: 5]
      - $\bowtie_b$ [tuples read in: 30 out: 25]
        - $\sigma_{b=0}$ [tuples read in: 25 out: 5]
          - R(a,b) has 25 tuples
        - $\sigma_{b=0}$ [tuples read in: 125 out: 25]
          - S(b,c) has 25 tuples
    - $\Pi_{['c']}$ [tuples read in: 25 out: 5]
      - $\sigma_{c=0}$ [tuples read in: 125 out: 25]
        - T(c,d) has 25 tuples

$\Pi_{a,c}((\Pi_c(\sigma_{c=0}(\text{T(c,d)}))) \bowtie_c ((\sigma_{b=0}(\sigma_{c=0}(\text{S(b,c)}))) \bowtie_b (\sigma_{b=0}(\text{R(a,b)}))))$

```
[(8, 0), (2, 0), (0, 0), (6, 0), (4, 0)]
True
```

Total Reads: 102

- $\Pi_{['a','c']}$ [tuples read in: 5 out: 5]
  - $\bowtie_c$ [tuples read in: 6 out: 5]
    - $\Pi_{['c']}$ [tuples read in: 5 out: 1]
      - $\sigma_{c=0}$ [tuples read in: 25 out: 5]
        - T(c,d) has 25 tuples
    - $\bowtie_b$ [tuples read in: 6 out: 5]
      - $\sigma_{b=0}$ [tuples read in: 5 out: 1]
        - $\sigma_{c=0}$ [tuples read in: 25 out: 5]
          - S(b,c) has 25 tuples
      - $\sigma_{b=0}$ [tuples read in: 25 out: 5]
        - R(a,b) has 25 tuples

关系代数优化

- 先选择，后连接
- 多个关系连接时，先数据量较小的两个关系的连接

$\sigma_{c=0}(\Pi_c(\sigma_{d=2}(\sigma_{a=3}((\ R(a,b)\ ) \bowtie_b (\ (\ S(b,c)\ ) \bowtie_c (\ T(c,d)\ ))))))$

[]

Total Reads: 18525

- $\sigma_{c=0}$ [tuples read in: 0 out: 0]
  - $\Pi_{['c']}$ [tuples read in: 0 out: 0]
    - $\sigma_{d=2}$ [tuples read in: 0 out: 0]
      - $\sigma_{a=3}$ [tuples read in: 375 out: 0]
        - $\bowtie_b$ [tuples read in: 1900 out: 375]
          - R(a,b) has 25 tuples
          - $\bowtie_c$ [tuples read in: 16250 out: 1875]
            - S(b,c) has 25 tuples
            - T(c,d) has 25 tuples

$\Pi_c((\ \sigma_{b=2}(\sigma_{a=3}(R(a,b)))\ ) \bowtie_b (\ (\ \sigma_{c=0}(\sigma_{b=2}(S(b,c)))\ ) \bowtie_c (\ \sigma_{c=0}(T(c,d))\ )))$

[]
True

Total Reads: 25

- $\Pi_{['c']}$ [tuples read in: 0 out: 0]
  - $\bowtie_b$ [tuples read in: 0 out: 0]
    - $\sigma_{b=2}$ [tuples read in: 0 out: 0]
      - $\sigma_{a=3}$ [tuples read in: 25 out: 0]
        - R(a,b) has 25 tuples
    - $\bowtie_c$ [tuples read in: 0 out: 0]
      - $\sigma_{c=0}$ [tuples read in: 0 out: 0]
        - $\sigma_{b=2}$ [tuples read in: 0 out: 0]
          - S(b,c) has 25 tuples
      - $\sigma_{c=0}$ [tuples read in: 0 out: 0]
        - T(c,d) has 25 tuples

空间索引

- create index countries_geom_idx on countries using gist(geom);
- create index rivers_geom_idx on rivers using gist(geom);
- create index cities_geom_idx on cities using gist(geom);

空间索引

- 查询每条河流穿越国家的次数，查询结果模式为(rivers.name, num)，按次数降序排列

```
explain analyze
select R.name, count(*)
from countries C, rivers R
where ST_Crosses(R.geom, C.geom)
group by R.gid, R.name
order by count(*) desc
```

空间索引

国家和河流的连接操作使用了哪个Join算法：
基于河流分组使用了什么算法实现：
按次数降序排列使用了什么算法实现：
按次数降序排列是在磁盘还是内存完成：

QUERY PLAN
Sort (cost=188868.69..188872.33 rows=1454 width=20) (actual time=188037.811..188037.831 rows=416 loops=1)
Sort Key: (count(*)) DESC
Sort Method: quicksort Memory: 56kB
-> GroupAggregate (cost=188729.03..188792.32 rows=1454 width=20) (actual time=188037.597..188037.753 rows=416 loops=1)
Group Key: r.gid
-> Sort (cost=188729.03..188745.28 rows=6500 width=12) (actual time=188037.591..188037.640 rows=793 loops=1)
Sort Key: r.gid
Sort Method: quicksort Memory: 64kB
-> Nested Loop (cost=0.00..188317.38 rows=6500 width=12) (actual time=11.992..188036.023 rows=793 loops=1)
Join Filter: ((r.geom && c.geom) AND _st_crosses(r.geom, c.geom))
Rows Removed by Join Filter: 369977
-> Seq Scan on countries c (cost=0.00..72.55 rows=255 width=34724) (actual time=0.008..0.147 rows=255 loops=1)
-> Seq Scan on rivers r (cost=0.00..356.54 rows=1454 width=2878) (actual time=0.001..0.315 rows=1454 loops=255)
Planning time: 0.175 ms
Execution time: 188037.890 ms

空间索引

- 查询规划利用了哪些空间索引：
- 国家和河流通过精确几何判断为穿越的行数与空间索引通过包围盒判断可能穿越的行数比值：793 / (793+18)
- 与2.1相比，查询规划估计的cost值(最大值)减少了(任意单位)：
- 与2.1相比，查询语句实际运行时间减少了(ms)：

QUERY PLAN
Sort (cost=1083.70..1087.34 rows=1454 width=20) (actual time=170488.835..170488.854 rows=416 loops=1)
Sort Key: (count(*)) DESC
Sort Method: quicksort Memory: 56kB
-> HashAggregate (cost=992.79..1007.33 rows=1454 width=20) (actual time=170488.723..170488.778 rows=416 loops=1)
Group Key: r.gid
-> Nested Loop (cost=0.14..960.29 rows=6500 width=12) (actual time=3.537..170486.211 rows=793 loops=1)
-> Seq Scan on countries c (cost=0.00..72.55 rows=255 width=34724) (actual time=0.009..0.317 rows=255 loops=1)
-> Index Scan using rivers_geom_idx on rivers r (cost=0.14..3.47 rows=1 width=2878) (actual time=147.910..668.503 rows=3 loops=255)
Index Cond: (geom && c.geom)
Filter: _st_crosses(geom, c.geom)
Rows Removed by Filter: 18
Planning time: 1.058 ms
Execution time: 170488.976 ms

8.8. Operators

&& — Returns TRUE if A's 2D bounding box intersects B's 2D bounding box.
&&(geometry,box2df) — Returns TRUE if a geometry's (cached) 2D bounding

空间索引

- 查询在亚马逊河流10个单位距离内的所有城市，查询结果模式为(cities.gid, cities.name) (使用ST_Distance实现)

```
explain analyze
select distinct cities.gid, cities.name
from rivers, cities
where rivers.name = 'Amazonas' and
ST_Distance(rivers.geom,cities.geom) <= 10
```

空间索引

- 查询规划估计的查询结果行数为：
- 查询语句实际的查询结果行数为：
- 结果去重时(distinct)采用了什么算法实现：

QUERY PLAN
HashAggregate (cost=2115.25..2164.20 rows=4895 width=13) (actual time=206.216..206.249 rows=273 loops=1)
Group Key: cities.gid, cities.name
-> Nested Loop (cost=0.00..2090.78 rows=4895 width=13) (actual time=14.580..205.986 rows=360 loops=1)
Join Filter: (st_distance(rivers.geom, cities.geom) <= '10'::double precision)
Rows Removed by Join Filter: 14326
-> Seq Scan on cities (cost=0.00..592.43 rows=7343 width=45) (actual time=0.010..1.420 rows=7343 loops=1)
-> Materialize (cost=0.00..360.19 rows=2 width=2866) (actual time=0.000..0.000 rows=2 loops=7343)
-> Seq Scan on rivers (cost=0.00..360.18 rows=2 width=2866) (actual time=0.113..0.313 rows=2 loops=1)
Filter: ((name)::text = 'Amazonas'::text)
Rows Removed by Filter: 1452
Planning time: 0.125 ms
Execution time: 206.300 ms

空间索引

- 查询在亚马逊河流10个单位距离内的所有城市，查询结果模式为(cities.gid, cities.name) (使用ST_DWithin实现)

```
explain analyze
select distinct cities.gid, cities.name
from rivers, cities
where rivers.name = 'Amazonas' and
ST_DWithin(cities.geom, rivers.geom, 10)
```

空间索引

- 查询规划利用了哪些空间索引：
- 河流和城市通过精确几何判断距离小于10的行数与空间索引通过包围盒判断距离可能小于10的行数比值：180 / (180 + 30)
- 与2.2相比，查询规划估计的cost值(最大值)减少了(任意单位)：
- 与2.2相比，查询语句实际运行时间减少了(ms)：206.3→12.276

QUERY PLAN
Nested Loop (cost=0.21..377.28 rows=1 width=2911) (actual time=0.167..12.225 rows=360 loops=1)
-> Seq Scan on rivers r (cost=0.00..360.18 rows=2 width=2870) (actual time=0.125..0.336 rows=2 loops=1)
Filter: ((name)::text = 'Amazonas'::text)
Rows Removed by Filter: 1452
-> Index Scan using cities_geom_idx on cities c (cost=0.21..8.54 rows=1 width=41) (actual time=0.127..5.912 rows=180 loops=2)
Index Cond: (geom && st_expand(r.geom, '10'::double precision))
Filter: ((r.geom && st_expand(geom, '10'::double precision)) AND _st_dwithin(geom, r.geom, '10'::double precision))
Rows Removed by Filter: 30
Planning time: 0.180 ms
Execution time: 12.276 ms

空间索引

- SET enable_indexscan = false;

QUERY PLAN
Nested Loop (cost=4.22..377.41 rows=1 width=2911) (actual time=0.257..11.982 rows=360 loops=1)
-> Seq Scan on rivers r (cost=0.00..360.18 rows=2 width=2870) (actual time=0.180..0.468 rows=2 loops=1)
Filter: ((name)::text = 'Amazonas'::text)
Rows Removed by Filter: 1452
-> Bitmap Heap Scan on cities c (cost=4.22..8.61 rows=1 width=41) (actual time=0.118..5.739 rows=180 loops=2)
Recheck Cond: (geom && st_expand(r.geom, '10'::double precision))
Filter: ((r.geom && st_expand(geom, '10'::double precision)) AND _st_dwithin(geom, r.geom, '10'::double precision))
Rows Removed by Filter: 30
Heap Blocks: exact=114
-> Bitmap Index Scan on cities_geom_idx (cost=0.00..4.22 rows=1 width=0) (actual time=0.067..0.067 rows=210 loops=2)
Index Cond: (geom && st_expand(r.geom, '10'::double precision))
Planning time: 0.184 ms
Execution time: 12.034 ms

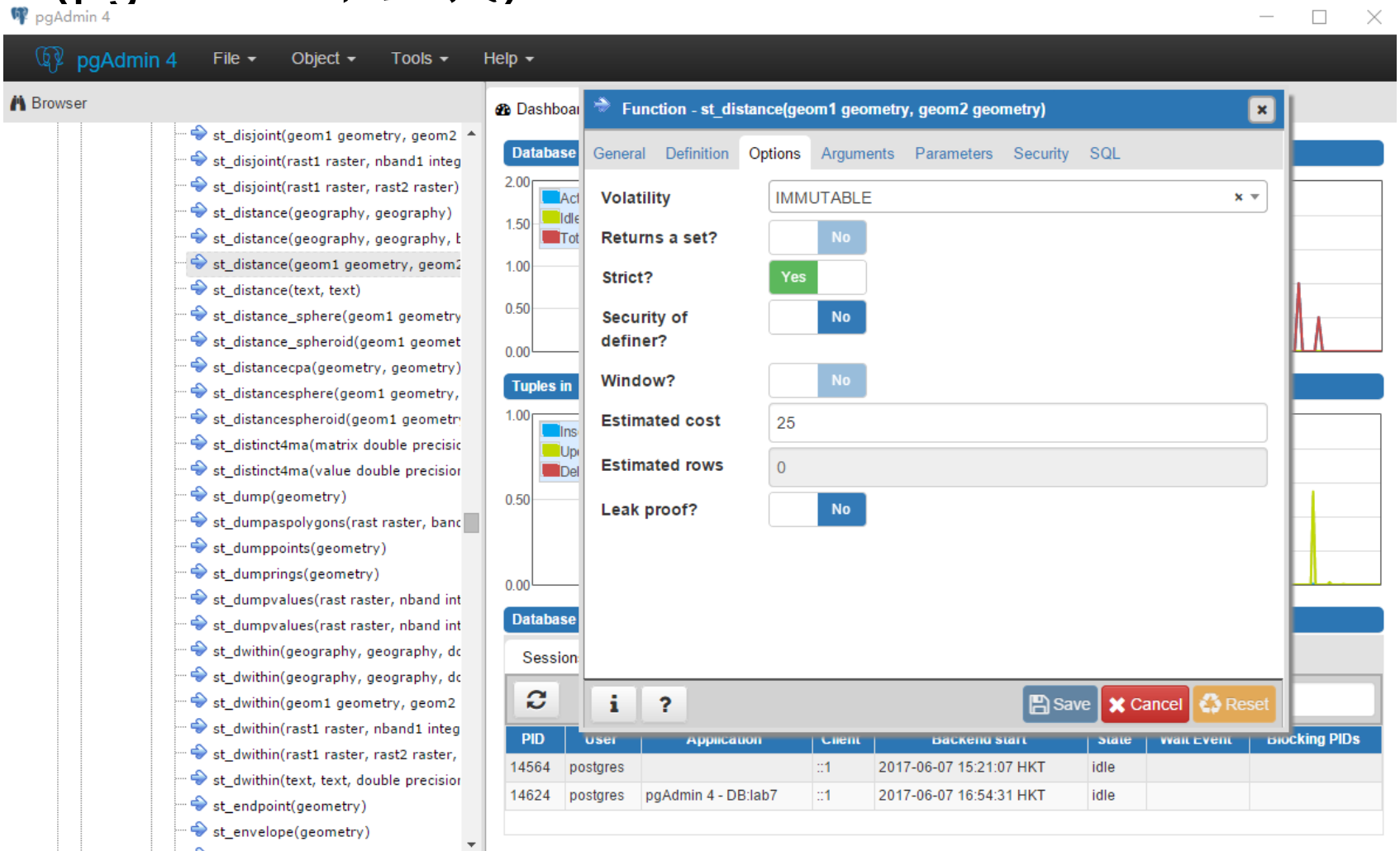
空间索引

- SET enable_indexscan = false;
- SET enable_bitmapscan = false;

QUERY PLAN
Nested Loop (cost=0.00..6716.86 rows=1 width=2911) (actual time=10.583..146.516 rows=360 loops=1)
Join Filter: ((c.geom && st_expand(r.geom, '10'::double precision)) AND (r.geom && st_expand(c.geom, '10'::double precision)) AND _st_dwithin(c.geom, r.geom, '10'::double precision))
Rows Removed by Join Filter: 14326
-> Seq Scan on cities c (cost=0.00..592.43 rows=7343 width=41) (actual time=0.012..1.070 rows=7343 loops=1)
-> Materialize (cost=0.00..360.19 rows=2 width=2870) (actual time=0.000..0.000 rows=2 loops=7343)
-> Seq Scan on rivers r (cost=0.00..360.18 rows=2 width=2870) (actual time=0.118..0.347 rows=2 loops=1)
Filter: ((name)::text = 'Amazonas'::text)
Rows Removed by Filter: 1452
Planning time: 0.164 ms
Execution time: 146.561 ms

空间索引

- ST_Distance(geometry, geometry)的cost值为(pgAdmin中查找)：



空间索引

- ST_DWithin(geometry, geometry)的cost值为(pgAdmin中查找)：