SQL Part 2

Leonardo Cunha, Matheus Centa

November 2019

1 Query Times

The execution time of the queries are as below:

Query	Time1 (ms)	Time 2 (ms)	Time3(ms)	Time4 (ms)	Time5 (ms)
1	2661	2650	2568	2712	2600
2	184	179	178	177	179
3	314	248	240	242	239
4	101	100	101	100	97
5	386	381	390	385	387
6	1090	1042	1045	1048	1042
7	8752	8796	9014	9217	9052
8	1318	1271	1318	1321	1346
9	527	530	526	529	565

2 Queries

2.1 Query 1

Query 1, as we've stated it, is below:

```
SELECT c.name as name, a.name as name
FROM country c, artist a
WHERE c.id = a.area
ORDER BY c.name, a.name;
```

The plan of query 1, as given by PostgreSQL is below:

```
Gather Merge(cost=78939.77..199555.39 rows=1033776 width=25)
 Workers Planned: 2
  Sort(cost=77939.74..79231.96 rows=516888 width=25)
  Sort Key: c.name, a.name
   Hash Join(cost=7.78..16519.38 rows=516888 width=25)
   Hash Cond: (a.area = c.id)
    Parallel Seq Scan on artist a(cost=0.00..15146.88 rows=516888 width=18)
```

```
    Hash(cost=4.57..4.57 rows=257 width=15)
     Seq Scan on country c(cost=0.00..4.57 rows=257 width=15)
```

As we can see, the complexity of the query is largely dominated by the sorting required by the ordering of the results. As it is a very straightforward query, we don't believe it's performance can be significantly optimized besides creating an index on the names of the countries/artists.

2.2 Query 2

Query 2, as we've stated it, is below:

```
SELECT c.name as name, a.name as name
FROM country c, artist a, (SELECT t.area
                           FROM (SELECT area, COUNT(*)
                                 FROM artist
                                 WHERE area IS NOT NULL
                                 GROUP BY area) as t
                           ORDER BY t.count DESC
                           LIMIT 1) as temp
WHERE c.id = temp.area AND a.area = temp.area;
```

The plan of query 2, as given by PostgreSQL is below:

```
Hash Join(cost=17122.49..44174.82 rows=4827 width=25)
Hash Cond: (a.area = c.id)
 Seq Scan on artist a  (cost=0.00..22383.32 rows=1240532 width=18)
  Hash(cost=17122.48..17122.48 rows=1 width=19)
    Merge Join(cost=17121.18..17122.48 rows=1 width=19)
    Merge Cond: (c.id = temp.area)
     Sort(cost=14.86..15.50 rows=257 width=15)
     Sort Key: c.id
      Seq Scan on country c(cost=0.00..4.57 rows=257 width=15)
       Sort(cost=17106.32..17106.33 rows=1 width=4)
       Sort Key: temp.area
        Subquery Scan on temp(cost=17106.30..17106.31 rows=1 width=4)
         Limit(cost=17106.30..17106.30 rows=1 width=12)
          Sort(cost=17106.30..17106.68 rows=154 width=12)
          Sort Key: (count(*)) DESC
           Finalize GroupAggregate(cost=17064.97..17103.99 rows=154 width=12)
           Group Key: artist.area
            Gather Merge(cost=17064.97..17100.91 rows=308 width=12)
            Workers Planned: 2
             Sort(cost=16064.95..16065.33 rows=154 width=12)
             Sort Key: artist.area
              Partial HashAggregate(cost=16057.81..16059.35 rows=154 width=12)
              Group Key: artist.area
```

```
Parallel Seq Scan on artist(cost=0.00..15146.88 rows=182186 width=4)
Filter: (area IS NOT NULL)
```

The complexity of query 3 is distributed between multiple operations, but the final join ends up being the most costly operation. We could easily optimize the query, given the conditions underlying the question by hard-coding the most prolific country, or by getting it as a result of a separate query and using it as view.

2.3 Query 3

Query 3, as we've stated it, is below:

```
SELECT DISTINCT a.id as id
FROM artist a, release_has_artist rha, release_country rc, country c
WHERE (rha.artist = a.id AND rha.release = rc.release
       AND CAST(rc.country AS INT) = c.id AND LEFT(c.name, 1) = 'A');
```

The plan of query 3, as given by PostgreSQL is below:

```
HashAggregate(cost=22725.01..22809.91 rows=8490 width=4)
Group Key: a.id
 Gather(cost=1006.72..22703.78 rows=8490 width=4)
 Workers Planned: 2
  Nested Loop(cost=6.72..20854.78 rows=3538 width=4)
   Nested Loop(cost=6.30..19188.57 rows=3538 width=4)
    Hash Join(cost=5.87..17908.39 rows=2574 width=4)
    Hash Cond: ((rc.country)::integer = c.id)
     Parallel Seq Scan on release_country rc(cost=0.00..15966.00 rows=661600 width=7)
      Hash(cost=5.86..5.86 rows=1 width=4)
       Seq Scan on country c(cost=0.00..5.86 rows=1 width=4)
       Filter: ("left"((name)::text, 1) = 'A'::text)
        Index Only Scan using release_has_artist_pkey on release_has_artist rha
        (cost=0.43..0.49 rows=1 width=8)
        Index Cond: (release = rc.release)
         Index Only Scan using artist_pkey on artist a
         (cost=0.43..0.47 rows=1 width=4)
         Index Cond: (id = rha.artist)
```

The costliest operation on this query is the sorting required by the "unique" condition. As with query 1, we don't think it can be significantly optimized by restating it.

2.4 Query 4

Query 4, as we've stated it, is below:

```
SELECT country as id, COUNT(*) as cc
FROM release_country
GROUP BY country
ORDER BY cc DESC;
```

The plan of query 4, as given by PostgreSQL is below:

```
Sort(cost=20317.39..20317.71 rows=130 width=11)
Sort Key: (count(*)) DESC
 Finalize GroupAggregate(cost=20279.89..20312.82 rows=130 width=11)
 Group Key: country
  Gather Merge(cost=20279.89..20310.22 rows=260 width=11)
  Workers Planned: 2
   Sort(cost=19279.86..19280.19 rows=130 width=11)
   Sort Key: country
    Partial HashAggregate(cost=19274.00..19275.30 rows=130 width=11)
    Group Key: country
     Parallel Seq Scan on release_country
     (cost=0.00..15966.00 rows=661600 width=3)
```

The cost of query 4 is well distributed over multiple operations (mainly the sorts and the gather merge) that are given as requirement in the exercise. Thus we don't think it can be optimized by reformulation.

2.5 Query 5

Query 5, as we've stated it, is below:

```
SELECT rha.release as release, rha.artist as artist
FROM release_has_artist rha
WHERE rha.contribution = 0;
```

The plan of query 5, as given by PostgreSQL is below:

```
Seq Scan on release_has_artist rha(cost=0.00..36935.90 rows=1841454 width=8)
Filter: (contribution = 0)
```

The only operation on query 5 is a sequential scan with a filter. Possibly creating an index on the 'contribution' column will result in a performance improvement.

2.6 Query 6

Query 6, as we've stated it, is below:

```
SELECT rha1.artist as artist, rha2.artist as artist, COUNT(*) as count
FROM release_has_artist rha1, release_has_artist rha2
WHERE rha1.artist <> rha2.artist AND rha1.release = rha2.release
GROUP BY rha1.artist, rha2.artist;
```

The plan of query 6, as given by PostgreSQL is below:

```
Finalize GroupAggregate(cost=211236.43..557881.51 rows=2870275 width=16)
Group Key: rha1.artist, rha2.artist
 Gather Merge(cost=211236.43..511239.54 rows=2391896 width=16)
 Workers Planned: 2
  Partial GroupAggregate(cost=210236.41..234155.37 rows=1195948 width=16)
  Group Key: rha1.artist, rha2.artist
   Sort(cost=210236.41..213226.28 rows=1195948 width=8)
   Sort Key: rha1.artist, rha2.artist
    Parallel Hash Join(cost=33847.68..73155.11 rows=1195948 width=8)
    Hash Cond: (rha1.release = rha2.release)
     Join Filter: (rha1.artist <> rha2.artist)
      Parallel Seq Scan on release_has_artist rha1
      (cost=0.00..19745.97 rows=859497 width=8)
       Parallel Hash(cost=19745.97..19745.97 rows=859497 width=8)
        Parallel Seq Scan on release_has_artist rha2
        (cost=0.00..19745.97 rows=859497 width=8)
```

The complexity of query 6 is dominated by the sorts corresponding to the GROUP BY statement. We don't think this query can be optimized through rewriting it though, as it is very simply stated.

2.7 Query 7

Query 7, as we've stated it, is below:

```
SELECT DISTINCT rc.country as country, rc.release as release
FROM (SELECT t.release, COUNT(t.release)
  FROM track t
  GROUP BY t.release) as aux, release_country as rc
WHERE aux.count > 1 AND aux.release = rc.release;
```

The plan of query 7, as given by PostgreSQL is below:

```
Unique(cost=1867934.95..1868780.71 rows=112769 width=7)
 Sort(cost=1867934.95..1868216.87 rows=112769 width=7)
 Sort Key: rc.country, rc.release
  Hash Join(cost=1816252.44..1858471.93 rows=112769 width=7)
  Hash Cond: (rc.release = aux.release)
   Seq Scan on release_country rc(cost=0.00..25228.40 rows=1587840 width=7)
    Hash(cost=1814502.84..1814502.84 rows=106608 width=4)
     Subquery Scan on aux(cost=1657482.71..1814502.84 rows=106608 width=4)
      Finalize GroupAggregate(cost=1657482.71..1813436.76 rows=106608 width=12)
      Group Key: t.release
      Filter: (count(t.release) > 1)
       Gather Merge(cost=1657482.71..1806240.69 rows=639650 width=12)
       Workers Planned: 2
```

```
Partial GroupAggregate(cost=1656482.69..1731409.19 rows=319825 width=12)
Group Key: t.release
 Sort(cost=1656482.69..1680392.10 rows=9563767 width=4)
 Sort Key: t.release
  Parallel Seq Scan on track t(cost=0.00..286091.67 rows=9563767 width=4)
```

The complexity of query 7 is distributed among several operations, but the sort corresponding to the DISTINCT ends up being the costliest. We could optimize it by eliminating it somehow.

2.8 Query 8

Query 8, as we've stated it, is below:

```
SELECT rc.release as release, rc.country as country
FROM release_country rc
WHERE rc.country IN (SELECT rc_after.country
                FROM release_country rc_before, release_country rc_after
                WHERE (rc_before.release = rc_after.release
                           AND (rc_before.year < rc_after.year
                                OR (rc_before.year = rc_after.year
                                AND rc_before.month < rc_after.month)
                                OR (rc_before.year = rc_after.year
                                AND rc_before.month = rc_after.month
                                AND rc_before.day < rc_after.day))));
```

The plan of query 8, as given by PostgreSQL is below:

```
Hash Join(cost=191313.45..238374.65 rows=1587840 width=7)
Hash Cond: ((rc.country)::text = (rc_after.country)::text)
 Seq Scan on release_country rc(cost=0.00..25228.40 rows=1587840 width=7)
  Hash(cost=191311.83..191311.83 rows=130 width=3)
   HashAggregate(cost=191310.53..191311.83 rows=130 width=3)
   Group Key: (rc_after.country)::text
    Merge Join(cost=0.85..189822.65 rows=595150 width=3)
    Merge Cond: (rc_before.release = rc_after.release)
    Join Filter: ((rc_before.year < rc_after.year)
                  OR ((rc_before.year = rc_after.year)
                  AND (rc_before.month < rc_after.month))
                  OR ((rc_before.year = rc_after.year)
                  AND (rc_before.month = rc_after.month)
                  AND (rc_before.day < rc_after.day)))
     Index Scan using release_country_pkey on release_country rc_before
     (cost=0.43..66690.36 rows=1587840 width=16)
      Materialize(cost=0.43..70659.96 rows=1587840 width=19)
       Index Scan using release_country_pkey on release_country rc_after
       (cost=0.43..66690.36 rows=1587840 width=19)
```

Query 8 complexity is dominated aswell by the many join operations. It could maybe be optimized by using union operations instead.

2.9 Query 9

Query 9, as we've stated it, is below:

```
CREATE VIEW acb AS (
  SELECT
    c.name AS country,
    a.name AS artist,
    concat(lpad(a.syear::text, 4, '0'::text), '-',
    lpad(a.smonth::text, 2, '0'::text), '-',
    lpad(a.sday::text, 2, '0'::text))::date
      AS borndate
  FROM artist a, country c
  WHERE
    c.id = a.area
    AND a.syear IS NOT NULL
    AND a.smonth IS NOT NULL
    AND a.sday IS NOT NULL
    AND a.type = 1);

SELECT
  globcount.country AS country,
  globcount.artist AS artist,
  loccount.nb_local AS nb,
  globcount.nb_global AS nb_global
FROM
  (SELECT
    country,
    artist,
    ((ROW_NUMBER() OVER (ORDER BY borndate))-1) AS nb_global
   FROM acb) AS globcount,
  (SELECT
    country,
    artist,
    ((ROW_NUMBER() OVER (PARTITION BY country ORDER BY borndate))-1) AS nb_local
   FROM acb) AS loccount
WHERE globcount.country = loccount.country AND globcount.artist = loccount.artist;

DROP VIEW acb;
```

The plan of query 9, as given by PostgreSQL is below:

```
Hash Join(cost=35530.29..35639.43 rows=1 width=41)
Hash Cond: (((c.name)::text = (globcount.country)::text)
```

```
          AND ((a.name)::text = (globcount.artist)::text))
WindowAgg(cost=17706.68..17788.53 rows=1559 width=37)
 Sort(cost=17706.68..17710.58 rows=1559 width=29)
 Sort Key: c.name, ((concat(lpad((a.syear)::text, 4, '0'::text), '-',
                    lpad((a.smonth)::text, 2, '0'::text), '-',
                    lpad((a.sday)::text, 2, '0'::text)))::date)
  Gather(cost=1007.78..17624.00 rows=1559 width=29)
  Workers Planned: 2
   Hash Join(cost=7.78..16468.10 rows=650 width=29)
   Hash Cond: (a.area = c.id)
    Parallel Seq Scan on artist a(cost=0.00..16439.10 rows=650 width=30)
    Filter: ((syear IS NOT NULL)
             AND (smonth IS NOT NULL)
             AND (sday IS NOT NULL)
             AND (type = 1))
     Hash(cost=4.57..4.57 rows=257 width=15)
      Seq Scan on country c  (cost=0.00..4.57 rows=257 width=15)
Hash(cost=17800.22..17800.22 rows=1559 width=33)
 Subquery Scan on globcount(cost=17706.68..17800.22 rows=1559 width=33)
  WindowAgg(cost=17706.68..17784.63 rows=1559 width=37)
   Sort(cost=17706.68..17710.58 rows=1559 width=29)
   Sort Key: ((concat(lpad((a_1.syear)::text, 4, '0'::text), '-',
              lpad((a_1.smonth)::text, 2, '0'::text), '-',
              lpad((a_1.sday)::text, 2, '0'::text)))::date)
    Gather(cost=1007.78..17624.00 rows=1559 width=29)
    Workers Planned: 2
     Hash Join(cost=7.78..16468.10 rows=650 width=29)
     Hash Cond: (a_1.area = c_1.id)
      Parallel Seq Scan on artist a_1(cost=0.00..16439.10 rows=650 width=30)
      Filter: ((syear IS NOT NULL)
               AND (smonth IS NOT NULL)
               AND (sday IS NOT NULL)
               AND (type = 1))
        Hash(cost=4.57..4.57 rows=257 width=15)
         Seq Scan on country c_1(cost=0.00..4.57 rows=257 width=15)
```

This query required quite a few optimizations just to return in a sensible time. The main optimization, though, was that the count of artists that were born before a given artist is just the the row number (minus one) of that artist when the group was sorted based on their starting dates in ascending fashion. We weren't able to perform any optimizations on this query.

3 Performance Improvement

The table below displays the performance of the optimized queries.

Query	Time1 (ms)	Time 2 (ms)	Time3(ms)	Time4 (ms)	Time5 (ms)
7	5552	5326	5323	5242	5573
8	908	856	844	889	869

3.1 Query 7

We chose to optimize the query 7 because it is one of the queries that took the longest time and because the query plan showed a promising way to improve on it. We restated the query as below:

```
SELECT rc.country AS country, rc.release AS release
FROM release_country AS rc
INNER JOIN (SELECT release
    FROM track
    GROUP BY release
            HAVING COUNT(*)>1) AS c ON rc.release = c.release;
```

The plan of the optimized query as given by PostgreSQL is below:

```
Hash Join(cost=1816252.44..1858471.93 rows=112769 width=7)
Hash Cond: (rc.release = track.release)
 Seq Scan on release_country rc(cost=0.00..25228.40 rows=1587840 width=7)
  Hash(cost=1814502.84..1814502.84 rows=106608 width=4)
   Finalize GroupAggregate  (cost=1657482.71..1813436.76 rows=106608 width=4)
   Group Key: track.release
   Filter: (count(*) > 1)
    Gather Merge(cost=1657482.71..1806240.69 rows=639650 width=12)
    Workers Planned: 2
     Partial GroupAggregate  (cost=1656482.69..1731409.19 rows=319825 width=12)
     Group Key: track.release
      Sort(cost=1656482.69..1680392.10 rows=9563767 width=4)
      Sort Key: track.release
       Parallel Seq Scan on track(cost=0.00..286091.67 rows=9563767 width=4)
```

3.2 Query 8

We chose to optimize the query 8 because a careful observation of the query plan led us to realize that we can eliminate the join operations and have a simpler and still correct query. Below is the reformulated query:

```
SELECT rc_after.release, rc_after.country
FROM release_country rc_before, release_country rc_after
WHERE (rc_before.release = rc_after.release
      AND (rc_before.year < rc_after.year
      OR (rc_before.year = rc_after.year AND rc_before.month < rc_after.month)
      OR (rc_before.year = rc_after.year AND rc_before.month = rc_after.month AND rc_bef
```

The plan of the optimized query as given by PostgreSQL is below:

```
Merge Join  (cost=0.85..189822.65 rows=595150 width=7)
Merge Cond: (rc_before.release = rc_after.release)
Join Filter: ((rc_before.year < rc_after.year)
    OR ((rc_before.year = rc_after.year)
    AND (rc_before.month < rc_after.month))
    OR ((rc_before.year = rc_after.year)
     AND (rc_before.month = rc_after.month)
     AND (rc_before.day < rc_after.day)))
 Index Scan using release_country_pkey on release_country rc_before
 (cost=0.43..66690.36 rows=1587840 width=16)
  Materialize(cost=0.43..70659.96 rows=1587840 width=19)
   Index Scan using release_country_pkey on release_country rc_after
   (cost=0.43..66690.36 rows=1587840 width=19)
```

As we can see, a much simpler formulation that eliminates the join operation.