Solutions for Exercise No. 11

23rd May 2017

1 Selection and Indexes

Given two SQL statements:

Provide the PostgreSQL execution plan for these queries and explain how the queries are executed and the reason why the strategies are chosen.

1. ```
   SELECT * FROM city WHERE Population > 10000000;
   ```

   The query plan is

	QUERY PLAN text
1	Seq Scan on city (cost=0.00..63.89 rows=1 width=40)
2	Filter: (population > 10000000::numeric)

   By default Postgres creates an index only on the primary key. Since attribute *Population* is not in the primary key no index on the attribute is available, and the sequential scan is just the only possibility.

2. ```
   SELECT AVG(Population) FROM city;
   ```

   The query plan is

	QUERY PLAN text
1	Aggregate (cost=63.89..63.90 rows=1 width=6)
2	-> Seq Scan on city (cost=0.00..56.11 rows=3111 width=6)

   The aggregation function is Average, that requires to know the population of all cities, thus, PostgreSQL performs the sequential scan in order to calculate the aggregation function.

Create the following index:

```
CREATE INDEX city_pop ON city USING BTREE (Population);
```

How does this statement affect the execution plans of both of the given queries? Explain your answer.

1. ```
   SELECT * FROM city WHERE Population > 10000000;
   ```

   The query plan has changed

	QUERY PLAN text
1	Index Scan using city_pop on city (cost=0.28..8.30 rows=1 width=40)
2	Index Cond: (population > 10000000::numeric)

   This is a selection with range predicate, and PostgreSQL estimates just one tuple in the result. Now the index on the attribute *Population* is available. It allows to calculate the query faster, by just traversing the tree until the leaf with the proper key is found.
   ```

2. `SELECT AVG(Population) FROM city;`

   Although, the index on *Population* is available now, Postgres uses sequential scan. The reason can be, that index is sparse and the information about number of duplicate values are not available, or the relation **city** is so small (fits within a single disk page), that the cost estimations are the same for plans with or without index usage.

   The ideal solution would be to scan the index and compute the AVG function without transferring the blocks with tuples from the relation.

2 Join Strategies

Assume relations **r** and **s** with schemas $R(A, B)$ and $S(C, D, E)$ and the following information:

- **r** has 100'000 tuples, **s** has 300'000 tuples.
- The size of a tuple in **r** is 20 bytes, the size of a tuple in **s** is 30 bytes.
- The disk block size is 4096 bytes.
- A tuple must be stored entirely in a block.
- The average time to retrieve a block from a disk is 2ms.
- **r** and **s** have separate buffers.
- The buffers have a size of one disk block.

Calculate the time for the block fetches of $r \bowtie_{A=C} s$ for the following join algorithms. State the formula you used and consider all possibilities to choose outer and inner relation.

First we calculate the number of blocks required to store each relation.
$b_r = \lceil \frac{100'000}{\lfloor \frac{4096}{20} \rfloor} \rceil = 491$ blocks.
$b_s = \lceil \frac{300'000}{\lfloor \frac{4096}{30} \rfloor} \rceil = 2'206$ blocks.
$n_r = 100'000$ tuples.
$n_s = 300'000$ tuples.

1. Block Nested Loop Join.

   r is inner and **s** is outer relation.

   The number of blocks to fetch is $b_r * b_s + b_s = 1'085'352$ blocks.

   The time for the block fetches is $2*(b_r*b_s+b_s) = 2*1'085'352 = 2'170'704$ ms.

   s is inner and **r** is outer relation.

   The number of blocks to fetch is $b_r * b_s + b_r = 1'083'637$ blocks.

   The time for the block fetches is $2*(b_r*b_s+b_r) = 2*1'083'637 = 2'167'274$ ms.

2. Indexed Nested Loop Join under the assumption that a primary B^+-tree index on attribute A with 3 levels is available.

   Since only the index on relation r is available, s is the outer relation.

   The number of blocks to fetch is $n_s * (3 + 1) + b_s = 1'202'206$ blocks.

   The time for the block fetches is $2 * (n_s * (3 + 1) + b_s) = 2'404'412$ ms.

3. Nested Loop Join with MRU under the assumption that the buffer size of the inner relation is 100 blocks and that the smaller relation is the outer one.

   r is outer and **s** is inner relation.

   (a) The simplified solution for the upper bound of number of fetched blocks:

   The number of blocks to fetch is $b_r + 1 * b_s + (n_r - 1) * (b_s - 99) = 491 + 2'206 + 99'999 * 2'107 = 210'700'590$ blocks.

   The time for the block fetches is $2 * (b_r + 1 * b_s + (n_r - 1) * (b_s - 99)) = 421'401'180$ ms.

   Explanation: **r** is fetched block by block to the buffer (b_r);

   for the first tuple of **r** all tuples from **s** must be fetched, thus, all blocks from **s** must be fetched for the first tuple ($1 * b_s$);

   for the rest of **r** tuples all blocks from **s** must be fetched except the 99 blocks, which are already in the buffer, because we always exchange the one most recent block in MRU ($(n_r - 1) * (b_s - 99)$).

   (b) The exact solution:

   For the exact solution we should count the number of cache hits for each of the tuples in r. We can make the following observations:

   - For each tuple except the first one there are at least 99 cache hits.
   - There are exactly 100 cache hits if there is already the first block of s in the buffer.
   - If the first block in the final state of the buffer just appeared for the first time after the gap, it will stay for the next 99 tuples.
   - After the first block disappears from the final state of the buffer it will not appear for the next 2106 blocks (to stay in the final state of the buffer first block prerequisites to have second block, then second block prerequisites the third one and so on).

   Thus, for the first tuple in r there is no cache hits, for the next 99 there are 100 cache hits, for the next 2106 tuples there are 99 cache hits, then again 100 hits for the next 99 tuples and so on. The period is 2205 (i.e., the blocks in the buffers after the 1^{st} tuple of r are the same as after the 2206^{th} tuple of r).

   Now we can calculate the exact number of fetched blocks. The number of full periods is $\lfloor \frac{100'000-1}{2'205} \rfloor = 45$, the last unfinished period is $100'000 - 1 - 2'205 * 45 = 774$, where for 99 tuples there are 100 hits and for the 675 tuples there are 99 hits.

The number of blocks to fetch is $b_r + 1 * b_s + (675 + 2'106 * 45) * (b_s - 99) + (99 + 99 * 45) * (b_s - 100) = 491 + 2'206 + 2'107 * 95'445 + 2'106 * 4'554 = 210'696'036$ blocks.

The time for the block fetches is $2 * 210'696'036 = 421'392'072$ ms.

4. Merge Sort Join under the assumption that **r** is sorted on A, **s** is sorted on C, and all tuples with equal join values are on the same block.

   Since the cost formula is symmetric to **r** and **s**, the time is the same for both cases, when **r** is inner and **s** is outer relation and vice versa.

   The number of blocks to fetch is $b_r + b_s = 2'697$ blocks.

   The time for the block fetches is $2 * (b_r + b_s) = 2 * 2'697 = 5'394$ ms.

5. Hash Join.

   Since the cost formula is symmetric to **r** and **s**, the time is the same for both cases, when **r** is inner and **s** is outer relation and vice versa.

   The number of blocks to fetch is $3 * (b_r + b_s) = 8'091$ blocks.

   The time for the block fetches is $2 * (3 * (b_r + b_s)) = 2 * 8'091 = 16'182$ ms.

3 Optimization

1. Given the following query:

```
SELECT Continent
FROM encompasses, country
WHERE country.Code = encompasses.Country;
```

   - Provide the PostgreSQL execution plan for this query and explain how the query is executed and the reason why this strategy is chosen.

     The query plan is

	QUERY PLAN text
1	Hash Join (cost=8.36..16.10 rows=242 width=8)
2	Hash Cond: ((encompasses.country)::text = (country.code)::text)
3	-> Seq Scan on encompasses (cost=0.00..4.42 rows=242 width=11)
4	-> Hash (cost=5.38..5.38 rows=238 width=3)
5	-> Seq Scan on country (cost=0.00..5.38 rows=238 width=3)

     This is an equi join with one comparison. Thus, hash join, merge join and nested join could be used.

     The hash join is the preferable option when the relations are not sorted on a join attribute. The relations are not sorted, thus the hash join is performed.

     The smaller relation **country** is a build input and **encompasses** is a probe input. The reason is the system assumes that the hash structure of the smaller relation can fit and be kept entirely in main memory.

     The relation **country** is scanned and the hash structure is computed for that relation. Then the system calculates the hash function of attribute *Country* for each tuple of **encompasses** and searches for the matching value in the hash structure.

- Disable the hash join with *set enable_hashjoin=off*. Give the new query plan and explain it, explain why it is slower than the previous plan.

  The query plan is

	QUERY PLAN text
1	Merge Join (cost=28.78..33.60 rows=242 width=8)
2	Merge Cond: ((country.code)::text = (encompasses.country)::text)
3	-> Sort (cost=14.77..15.37 rows=238 width=3)
4	Sort Key: country.code
5	-> Seq Scan on country (cost=0.00..5.38 rows=238 width=3)
6	-> Sort (cost=14.00..14.61 rows=242 width=11)
7	Sort Key: encompasses.country
8	-> Seq Scan on encompasses (cost=0.00..4.42 rows=242 width=11)

  The system sorts the input relations and then performs the Merge Scan and joins them.

  The cost estimation for this plan is 33.60, that is slower than 16.10, which is the cost estimation for the previous plan. But since the hash join is disabled, thus, the sort merge join is performed.

  This strategy is still faster than nested loop join, but since the relations are not sorted, sorting creates the overhead of $O(B_{country}logB_{country}+B_{encompasses}logB_{encompasses})$.

- Disable the merge join with *set enable_mergejoin=off*. Give the new query plan and explain it, explain why it is slower than the previous plan.

  The query plan is

	QUERY PLAN text
1	Nested Loop (cost=0.14..63.03 rows=242 width=8)
2	-> Seq Scan on country (cost=0.00..5.38 rows=238 width=3)
3	-> Index Only Scan using encompasseskey on encompasses (cost=0.14..0.23 rows=1 width=11)
4	Index Cond: (country = (country.code)::text)

  The last used approach is the Indexed Nested Loop Join. The index on the key attribute *Country* of **encompasses** is available since PostgreSQL creates an index by default on any primary key. This plan has the worst estimation cost of 63.03 among all considered plans. It is slower, because for each outer tuple, the index must be traversed for fetching the matching tuples.

- Suggest a better strategy of the query execution, under the assumption that the attribute *Country* in relation **encompasses** is a foreign key, that references **country**(*Code*).

  The sequential scan of the relation **encompasses**, that returns only the *Continent*.

  The attribute *Code* is a primary key in the relation **country** and the attribute *Country* in relation **encompasses** is a foreign key to the attribute *Code* and a part of the primary key, and thus has only non-null values, which are the same as in *Code*. Thus, for each tuple in **encompasses** the join predicate (*country.Code = encompasses.Country*) has exactly one matching tuple in **country**. Attribute *Continent* of the relation **encompasses** is returned, thus, only relation **encompasses** should be scanned.

2. Given the following query:

```
SELECT DISTINCT Code
FROM country;
```

- Provide the PostgreSQL execution plan for this query and explain how the query is executed.

  The query plan is

	QUERY PLAN text
1	HashAggregate (cost=5.97..8.36 rows=238 width=3)
2	Group Key: code
3	-> Seq Scan on country (cost=0.00..5.38 rows=238 width=3)

  In order to return all unique codes the relation **country** is scanned, and the grouping over the attribute *Code* using a temporal hash table is performed.

  Hashing or sorting can be performed in order to eliminate duplicates in the result.

  With disabled hashagg (*set enable_hashagg=off*) the query plan is

	QUERY PLAN text
1	Unique (cost=14.77..15.96 rows=238 width=3)
2	-> Sort (cost=14.77..15.37 rows=238 width=3)
3	Sort Key: code
4	-> Seq Scan on country (cost=0.00..5.38 rows=238 width=3)

  PostgreSQL estimates the lower cost for creating the hash structure (8.36) rather than sorting the relation (15.37). The reason might be that the relation is small and the entire hash structure can fit into the main memory, which would not be possible for large relations.

- Suggest a better strategy of the query execution.

  The attribute *Code* is a primary key of the relation **country**. Each tuple in this relation has a unique value in *Code*, and the key word **DISTINCT** in the query does not affect the result. Thus, the third approach is possible for this query. The sequential scan would be the better strategy to perform this query.