EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

DB2
Sommersemester 2022
Prof. Torsten Grust, Denis Hirn
WSI — Database Systems Research Group

Assignment 10

Handin until: Friday, 22.07.2022, 09:00

1. [22 Points] **Join Operators in *PostgreSQL***

   Important Note: Before you start working on this assignment, **disable parallelism**:

```
set max_parallel_workers_per_gather = 0;
set max_parallel_workers = 0;
```

   Nested Loop Join: **Disable Hash- and Merge Join to answer the following questions!**

```
set enable_hashjoin = off;
set enable_mergejoin = off;
```

   (a) Create tables `one` and `many` as provided in `one-many.sql`. As long as no index is defined for any table, a query to join the tables will use a simple *Nested Loop Join* and will not terminate in an reasonable amount of time. Let's disable materialization first with

```
set enable_material = off;
```

   and use `EXPLAIN` (without `ANALYZE`) to show the most naive plan for the following query `Q`:

```
SELECT *
FROM   one AS o, many AS m
WHERE  o.a = m.a
```

   Based on the estimated `rows`, how often would the *Join Filter* `o.a = m.a` be evaluated?

   (b) Re-enable *materialization* and compare the new plan for query `Q` with (a).

```
set enable_material = on;
```

   Explain why the loop order has changed and why `Materialize` is used on the `Seq Scan` of `one`, instead of `many`.

   (c) A `PRIMARY KEY` index on `one(a)` supports the *Nested Loop Join* on query `Q`. How?

```
ALTER TABLE one ADD CONSTRAINT one_a PRIMARY KEY (a);
ANALYZE;
```

   Show the plan with `EXPLAIN ANALYZE` and explain it briefly.

   (d) An additional `PRIMARY KEY` index on `many(a,c)` further improves the query performance.

```
ALTER TABLE many ADD CONSTRAINT many_a_c PRIMARY KEY (a,c);
ANALYZE;
```

   - Why is only one of the two indexes used, while the other table is accessed using a `Seq Scan`?
   - Why is table `many` with index `many_a_c` (and not table `one` with `one_a`) preferred here as an inner join table?

   (e) How does the following change to the query `Q` benefit from both indexes instead?

```
SELECT *
FROM   one AS o, many AS m
WHERE  o.a = m.a
ORDER BY m.a
```

Hash Join: Re-enable Hash- and Merge Join to answer the following questions!

```
set enable_hashjoin = on;
set enable_mergejoin = on;
```

(f) If available, a *Hash Join* is used to answer the equi-join query `Q`. Show the plan with `EXPLAIN (VERBOSE, ANALYZE, BUFFERS)`.
- Why is table `one` (and not table `many`) chosen as the inner *build table*?
- Why are the indexes `one_a` and `many_a_c` not used to access the base tables?

(g) *Hash Join* builds up a temporary hash table and therefore suffers when `work_mem` is reduced. Issue `set work_mem='64kB'` (instead of default `'4MB'`) and re-execute query `Q`. Use the output of `EXPLAIN (VERBOSE, ANALYZE, BUFFERS)` to compare it with (f). Why does the performance of the *Hash Join* decrease significantly?

(h) Since even with low `work_mem`, some of the hash table is stored in-memory, the actual performance of (g) may depend heavily on the data distribution of the table being searched. Table `many_skewed` in `one-many.sql` provides a variant of table `many` with a highly skewed distribution of column `a`.

Examine columns `n_distinct`, `most_common_vals` and `most_common_freqs` in table `pg_stats`[1] to show statistics about the distribution of values for both, attribute `a` in `many` and `a` in `many_skewed`.
- Give a short comparison.

Execute the following query on `work_mem='64kB'` and compare its plan to (g).
Note: You may have to execute each query twice to avoid inconsistencies in caching.

```
SELECT *
FROM   one AS o, many_skewed AS m
WHERE  o.a = m.a
```

The I/O on temporary tables is reduced. To which number and why?

Merge Join: Disable Hash Join to answer the following question!

```
set enable_hashjoin = off;
```

(i) When we enforce *Merge Join* in (g) (`Q` with low `work_mem` of `'64kB'`), it performs better than the original *Hash Join*.
- How does the *Merge Join* make use of indexes `one_a` and `many_a_c`?
- Why is memory no crucial factor here, so that *Merge Join* can outperform the *Hash Join* on `work_mem='64kB'`?

2. [8 Points] **External Merge Sort**

Given an input table of unsorted values distributed over 16 pages. Each page covers up to two elements:

pages 1–8:	38,58	23,31	36,59	21,35	19,53	13,25	56,22	54,21
pages 9–16:	34,58	19,14	48,46	47,32	60,40	58,60	55,24	50,54

Sort the table using **External Merge Sort**, with an available working memory of $B = 3$ **pages**.
For each *Pass* of the algorithm, write down the *output runs* written to secondary memory.

Note: *Pass 0* does **not** use *Replacement Sort* and returns sorted runs of size `B` pages.

[1] https://www.postgresql.org/docs/current/static/view-pg-stats.html