Prof. Dr. T. Grust, Benjamin Dietrich, Denis Hirn SS 2020

DB2

Forum: https://forum-db.informatik.uni-tuebingen.de/c/ss20-db2

Assignment 11 (14.7.2020)

Submission: Tuesday, 21.7.2020, 10:00 AM

Relevant videos: up to `DB2 - Chapter 14 - Video #83`.

https://tinyurl.com/DB2-2020

1. [20 Points] **Join Operators in *PostgreSQL***

Important Note: Before you start following the instructions of this assignment, **disable parallelism** to always see the relevant query plans:

```
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 there is no index defined on any table, a query to join them makes use of simple *Nested Loop Join* and will not terminate in reasonable time. First let us switch off materialization 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) Switch *materialization* on again and compare the new plan for query Q to 1a.

```
set enable_material = on;
```

- Can you think of a reason why the loop order has changed and `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 using `EXPLAIN ANALYZE` and explain 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 both 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 as inner join table here?

(e) How does the following modification of 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 using `EXPLAIN (VERBOSE, ANALYZE, BUFFERS)`.

- Why is table `one` (and not table `many`) chosen as the inner *build table* here?
- Why are the indexes `one_a` and `many_a_c` not used to access the base tables here?

(g) *Hash Join* builds a temporary hash table and thus 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 1f.

- The *Hash Join* performance decreases significantly. Why?

(h) Since, even for low `work_mem`, a part of the hash table is stored in-memory, the actual performance of 1g can highly depend on the data distribution of the probed table. Table `many_skewed` in `one-many.sql` provides a variant of table `many` with a heavily skewed distribution on 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 1g.
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 1g (Q with low `work_mem` of `'64kB'`), we can observe that it performs quite 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'`?

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