Assignment 5

Join Tuning

Database Tuning

A4
Balint Peter, 12213073
Günter Lukas, 12125639
Ottino David, 51841010

June 2, 2025

Experimental Setup

Describe your experimental setup in a few lines.

We use a Python-script to run all of the tests in one go, measure the execution times and return the needed values. The queries are prepared once and the additional statements are added for each subsection of the tests to create indexes and force Postgres to use specific joins. We then drop all old indexes and run the subsection of queries prepared for each test.

Join Strategies Proposed by System

Response times

Indexes	Join Strategy Q1	Join Strategy Q2
no index	Hash Join (2.7250s)	Parallel Hash Join (0.3035s)
unique non-clustering on `Publ.pubID`	Hash Join (2.6926s)	Indexed Nested Loop Join (0.1715s)
clustering on `Publ.pubID` and `Auth.pubID`	Merge Join (2.4625s)	Indexed Nested Loop Join (0.2392s)

Discussion Discuss your observations. Is the choice of the strategy expected? How does the system come to this choice?

Query 1

With no indexes at all, PostgreSQL chooses to perform a hash join on Publ.pubID (the side with the smaller number of tuples); this makes sense, since hash joins are more performant than nested loop joins without indexes when both sides have a large amount of tuples.

This does not change with a non-clustering index on Publ. This is probably because the number of hash lookups is significantly smaller in this case than the number of index lookups that would be required when using the non-clustering index.

Finally, when both sides have a clustering index each, the database decides to run a merge join using the two indexes. Evidently though, this is not much more performant than the two alternatives; indeed, when performing the experiment on an another system, PostgreSQL ended up using a hash join instead.

Query 2

For the second query without an index, the optimizer starts a parallel sequential scan on Auth, filling up the hash table with entries where the name matches. This is only possible here because the required hash table is limited in size by the name filter; in a parallel hash, the processes cooperate to create a huge hash table, which is not realistically possible with millions of entries.

With a non-clustering index on Publ, we can also see a change: here, the optimizer can finally use the index, since the tuples in Auth are greatly filtered down according to the 'name' column.

The strategy does not change with the two clustered indexes. A merge join would be possible here, but it is not used, presumably because it would require a separate filter and sort step on the Auth index before the scan on Publ could be started.

Indexed Nested Loop Join

Response times

Indexes	Response time Q1 [s]	Response time Q2 [s]
index on `Publ.pubID`	13.6798	0.1692
index on `Auth.pubID`	5.7032	4.8619
index on `Publ.pubID` and `Auth.pubID`	5.3769	0.1867

Query plans

Index on `Publ.pubID` (Q1/Q2):

```
Creating non-clustered index on Publ.pubID
Gather  (cost=1000.43..987024.16 rows=3095200 width=82) (actual time=0.537..12599.057 rows=309519
  Workers Planned: 2
  Workers Launched: 2
  ->  Nested Loop  (cost=0.43..676504.16 rows=1289667 width=82) (actual time=1.060..12261.941 row
        ->  Parallel Seq Scan on auth  (cost=0.00..39705.67 rows=1289667 width=38) (actual time=0
        ->  Index Scan using idx_publ_pubid_nonclustered on publ  (cost=0.43..0.48 rows=1 width=8
              Index Cond: ((pubid)::text = (auth.pubid)::text)

Creating non-clustered index on Publ.pubID
Gather  (cost=1000.43..44016.78 rows=24 width=67) (actual time=36.762..173.169 rows=183 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Nested Loop  (cost=0.43..43014.38 rows=10 width=67) (actual time=18.211..119.609 rows=61 lo
        ->  Parallel Seq Scan on auth  (cost=0.00..42929.83 rows=10 width=23) (actual time=17.422
              Filter: ((name)::text = 'Divesh Srivastava'::text)
              Rows Removed by Filter: 1031672
        ->  Index Scan using idx_publ_pubid_nonclustered on publ  (cost=0.43..8.45 rows=1 width=8
              Index Cond: ((pubid)::text = (auth.pubid)::text)
```

Index on `Auth.pubID` (Q1/Q2):

```
Creating non-clustered index on Auth.pubID
Gather  (cost=1000.43..662542.57 rows=3095200 width=82) (actual time=0.563..5169.208 rows=309519
  Workers Planned: 2
  Workers Launched: 2
  ->  Nested Loop  (cost=0.43..352022.57 rows=1289667 width=82) (actual time=1.027..4927.963 rows
        ->  Parallel Seq Scan on publ  (cost=0.00..28618.39 rows=513839 width=89) (actual time=0.
        ->  Index Scan using idx_auth_pubid_nonclustered on auth  (cost=0.43..0.60 rows=3 width=3
              Index Cond: ((pubid)::text = (publ.pubid)::text)

Creating non-clustered index on Auth.pubID
Gather  (cost=1000.43..346601.98 rows=24 width=67) (actual time=399.018..4802.060 rows=183 loops=
  Workers Planned: 2
  Workers Launched: 2
  ->  Nested Loop  (cost=0.43..345599.58 rows=10 width=67) (actual time=359.002..4745.602 rows=61
        ->  Parallel Seq Scan on publ  (cost=0.00..28618.39 rows=513839 width=89) (actual time=0.
        ->  Index Scan using idx_auth_pubid_nonclustered on auth  (cost=0.43..0.61 rows=1 width=2
              Index Cond: ((pubid)::text = (publ.pubid)::text)
              Filter: ((name)::text = 'Divesh Srivastava'::text)
              Rows Removed by Filter: 3
```

Index on `Auth.pubID` and `Auth.pubID` (Q1/Q2):

```
Creating non-clustered index on Publ.pubID
Creating non-clustered index on Auth.pubID
Gather  (cost=1000.43..662542.57 rows=3095200 width=82) (actual time=0.514..4953.691 rows=309519
  Workers Planned: 2
  Workers Launched: 2
  ->  Nested Loop  (cost=0.43..352022.57 rows=1289667 width=82) (actual time=1.034..4714.075 rows
        ->  Parallel Seq Scan on publ  (cost=0.00..28618.39 rows=513839 width=89) (actual time=0.
        ->  Index Scan using idx_auth_pubid_nonclustered on auth  (cost=0.43..0.60 rows=3 width=3
              Index Cond: ((pubid)::text = (publ.pubid)::text)

Creating non-clustered index on Publ.pubID
Creating non-clustered index on Auth.pubID
Gather  (cost=1000.43..44016.78 rows=24 width=67) (actual time=45.347..188.893 rows=183 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Nested Loop  (cost=0.43..43014.38 rows=10 width=67) (actual time=17.758..130.616 rows=61 lo
        ->  Parallel Seq Scan on auth  (cost=0.00..42929.83 rows=10 width=23) (actual time=16.926
              Filter: ((name)::text = 'Divesh Srivastava'::text)
              Rows Removed by Filter: 1031672
        ->  Index Scan using idx_publ_pubid_nonclustered on publ  (cost=0.43..8.45 rows=1 width=8
              Index Cond: ((pubid)::text = (auth.pubid)::text)
```

Discussion Discuss your observations. Are the response times expected? Why (not)?

Query 1

When using a non-clustering index on Publ.pubID, the database performs a sequential scan (which results in the longest runtime of all experiments) and accesses the tuples from the Publ table based on the index. In contrast, with an index on Auth.pubID, Publ is read sequentially, while the index is used for Auth.

As Auth with 3 million tuples is significantly larger than Publ with 1.3 million, it is generally more efficient to scan Publ completely. This assumption is confirmed by the shorter runtime in the corresponding case.

If both tables are indexed, the system uses the index on Auth.pubID, while the index

on Publ is ignored. The runtime is not noticeably different from the variant in which only Auth.pubID is indexed - this explains the optimizer's decision.

Query 2

If a non-clustering index is used on Publ.pubID, PostgreSQL runs through the Auth table sequentially and filters for rows with *name = 'Divesh Srivastava'*. The corresponding tuple from Publ can then be found very efficiently using the index via pubID - as this column is unique. This combination leads to a significantly better runtime compared to query 1.

On the other hand, if there is only one index on Auth.pubID, PostgreSQL decides to use the smaller Publ table as the outer table to perform an indexed nested loop join, whereby the matching tuple from Auth is searched for each row in Publ by index. The name filter on Auth is applied after the index access. Here too, the use of the index ensures a better runtime, although not quite as efficiently as in the previous case.

If both columns (Publ.pubID and Auth.pubID) are indexed, PostgreSQL only uses the index on Publ.pubID in our case. The additional index on Auth remains unused. The execution time is therefore not noticeably different from the case in which only Publ.pubID is indexed.

Sort-Merge Join

Response times

Indexes	Response time Q1 [s]	Response time Q2 [s]
no index	8.9741	1.4914
two non-clustering indexes	2.4118	0.7319
two clustering indexes	2.4007	0.7481

Query plans

No index (Q1/Q2):

```
Gather  (cost=529161.49..867548.07 rows=3095200 width=82) (actual time=3735.644..6898.867 rows=30
  Workers Planned: 2
  Workers Launched: 2
  ->  Merge Join  (cost=528161.49..557028.07 rows=1289667 width=82) (actual time=3619.728..6181.4
        Merge Cond: ((auth.pubid)::text = (publ.pubid)::text)
        ->  Sort  (cost=241129.63..244353.80 rows=1289667 width=38) (actual time=1292.754..1630.5
              Sort Key: auth.pubid
              Sort Method: external merge  Disk: 54376kB
              Worker 0:  Sort Method: external merge  Disk: 47200kB
              Worker 1:  Sort Method: external merge  Disk: 46856kB
              ->  Parallel Seq Scan on auth  (cost=0.00..39705.67 rows=1289667 width=38) (actual
        ->  Materialize  (cost=287031.23..293197.30 rows=1233213 width=89) (actual time=2268.571.
              ->  Sort  (cost=287031.23..290114.26 rows=1233213 width=89) (actual time=2268.552..
                    Sort Key: publ.pubid
                    Sort Method: external merge  Disk: 121608kB
                    Worker 0:  Sort Method: external merge  Disk: 121608kB
                    Worker 1:  Sort Method: external merge  Disk: 121608kB
                    ->  Seq Scan on publ  (cost=0.00..35812.13 rows=1233213 width=89) (actual tim

Gather  (cost=170202.78..172771.80 rows=24 width=67) (actual time=1354.606..1675.787 rows=183 loo
  Workers Planned: 2
```

```
  Workers Launched: 2
  ->  Merge Join  (cost=169202.78..171769.40 rows=10 width=67) (actual time=945.614..1362.222 row
        Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
        ->  Sort  (cost=103702.98..104987.57 rows=513839 width=89) (actual time=509.776..796.875
              Sort Key: publ.pubid
              Sort Method: external merge  Disk: 47152kB
              Worker 0:  Sort Method: external merge  Disk: 38544kB
              Worker 1:  Sort Method: external merge  Disk: 35984kB
              ->  Parallel Seq Scan on publ  (cost=0.00..28618.39 rows=513839 width=89) (actual t
        ->  Sort  (cost=65499.55..65499.61 rows=24 width=23) (actual time=393.460..393.482 rows=1
              Sort Key: auth.pubid
              Sort Method: quicksort  Memory: 25kB
              Worker 0:  Sort Method: quicksort  Memory: 25kB
              Worker 1:  Sort Method: quicksort  Memory: 25kB
              ->  Seq Scan on auth  (cost=0.00..65499.00 rows=24 width=23) (actual time=31.243..3
                    Filter: ((name)::text = 'Divesh Srivastava'::text)
                    Rows Removed by Filter: 3095017
```

Two non-clustering indexes (Q1/Q2):

```
Creating non-clustered index on Publ.pubID
Creating non-clustered index on Auth.pubID
Merge Join  (cost=0.86..235103.52 rows=3095200 width=82) (actual time=0.018..2937.838 rows=309519
  Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
  ->  Index Scan using idx_publ_pubid_nonclustered on publ  (cost=0.43..74420.80 rows=1233213 wid
  ->  Index Scan using idx_auth_pubid_nonclustered on auth  (cost=0.43..118986.49 rows=3095200 wi

Creating non-clustered index on Publ.pubID
Creating non-clustered index on Auth.pubID
Merge Join  (cost=43933.21..121360.18 rows=24 width=67) (actual time=258.439..1137.923 rows=183 l
  Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
  ->  Index Scan using idx_publ_pubid_nonclustered on publ  (cost=0.43..74420.80 rows=1233213 wid
  ->  Sort  (cost=43932.78..43932.84 rows=24 width=23) (actual time=184.412..184.549 rows=183 loo
        Sort Key: auth.pubid
        Sort Method: quicksort  Memory: 25kB
        ->  Gather  (cost=1000.00..43932.23 rows=24 width=23) (actual time=34.355..183.927 rows=1
              Workers Planned: 2
              Workers Launched: 2
              ->  Parallel Seq Scan on auth  (cost=0.00..42929.83 rows=10 width=23) (actual time=
                    Filter: ((name)::text = 'Divesh Srivastava'::text)
                    Rows Removed by Filter: 1031672
```

Two clustering indexes (Q1/Q2):

```
Creating clustered index on Publ.pubID
Creating clustered index on Auth.pubID
Merge Join  (cost=0.86..235103.52 rows=3095200 width=82) (actual time=0.033..3320.097 rows=309519
  Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
  ->  Index Scan using idx_publ_pubid_clustered on publ  (cost=0.43..74420.80 rows=1233213 width=
  ->  Index Scan using idx_auth_pubid_clustered on auth  (cost=0.43..118986.49 rows=3095200 width

Creating clustered index on Publ.pubID
Creating clustered index on Auth.pubID
Merge Join  (cost=43933.21..121360.18 rows=24 width=67) (actual time=320.572..1376.520 rows=183 l
  Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
  ->  Index Scan using idx_publ_pubid_clustered on publ  (cost=0.43..74420.80 rows=1233213 width=
  ->  Sort  (cost=43932.78..43932.84 rows=24 width=23) (actual time=226.262..226.388 rows=183 loo
        Sort Key: auth.pubid
```

```
Sort Method: quicksort  Memory: 25kB
->  Gather  (cost=1000.00..43932.23 rows=24 width=23) (actual time=52.396..225.620 rows=1
      Workers Planned: 2
      Workers Launched: 2
      ->  Parallel Seq Scan on auth  (cost=0.00..42929.83 rows=10 width=23) (actual time=
            Filter: ((name)::text = 'Divesh Srivastava'::text)
            Rows Removed by Filter: 1031672
```

Discussion Discuss your observations. Are the response times expected? Why (not)?

Query 1

When using a merge join, the presence of non-clustered indexes on Publ.pubID and Auth.pubID has no influence on the runtime. In both cases, the tables must first be explicitly sorted according to the joined column before the merge join can be executed. Non-clustered indexes offer no advantage - neither during sorting nor during the subsequent join.

The situation is different if there are clustered indexes on the joined column in both tables. In this case, the data is already sorted so that the sorting effort is completely eliminated. This results in a significantly lower runtime, as the merge join can be executed directly on the pre-sorted data.

Query 2

The second query exhibits similar behavior: if there is no clustering index on the join columns Publ.pubID and `Auth.pubID`, both tables must first be explicitly sorted before the merge join can be performed. The only difference to the first query is that Auth is filtered to *name = 'Divesh Srivastava'* before sorting. This reduces the amount of tuples joined, leading to a shorter runtime.

If both columns have a clustering index, Auth is also filtered by name first. As Publ is already sorted based on the clustering index, only Auth needs to be sorted. In this case, the runtime becomes significantly shorter, as the join can be performed with barely any sorting effort.

Hash Join

Response times

Indexes	Response time Q1 [ms]	Response time [ms] Q2
no index	2.6734	0.3059

Query plans

No Index (Q1/Q2):

```
Hash Join  (cost=69292.29..236041.29 rows=3095200 width=82) (actual time=830.148..4453.022 rows=3
  Hash Cond: ((auth.pubid)::text = (publ.pubid)::text)
  ->  Seq Scan on auth  (cost=0.00..57761.00 rows=3095200 width=38) (actual time=0.068..454.953 r
  ->  Hash  (cost=35812.13..35812.13 rows=1233213 width=89) (actual time=827.562..827.563 rows=12
        Buckets: 65536  Batches: 32  Memory Usage: 5111kB
        ->  Seq Scan on publ  (cost=0.00..35812.13 rows=1233213 width=89) (actual time=0.014..252

Gather  (cost=43929.96..80258.37 rows=24 width=67) (actual time=226.941..344.091 rows=183 loops=1
  Workers Planned: 2
```

```
Workers Launched: 2
->  Parallel Hash Join  (cost=42929.96..79255.97 rows=10 width=67) (actual time=169.703..277.42
      Hash Cond: ((publ.pubid)::text = (auth.pubid)::text)
      ->  Parallel Seq Scan on publ  (cost=0.00..28618.39 rows=513839 width=89) (actual time=0.
      ->  Parallel Hash  (cost=42929.83..42929.83 rows=10 width=23) (actual time=156.991..156.9
            Buckets: 1024  Batches: 1  Memory Usage: 104kB
            ->  Parallel Seq Scan on auth  (cost=0.00..42929.83 rows=10 width=23) (actual time=
                  Filter: ((name)::text = 'Divesh Srivastava'::text)
                  Rows Removed by Filter: 1031672
```

Discussion What do you think about the response time of the hash index vs. the response times of sort-merge and index nested loop join for each of the queries? Explain.

Overall, we can observe that without indexes, a hash join typically achieves the best runtime. With indexes, the optimal join strategy strongly depends on whether the tables are clustered after the column to be joined, and whether the number of tuples to be processed can be restricted before the join.

Query 1

No selection is done in the first query - all tuples from Auth and Publ must be joined. In this case, the sort-merge join ends up being the most efficient strategy, provided that both tables are clustered on the join columns. The existing sorting eliminates additional effort.

However, without clustering, PostgreSQL must sort both tables before the merge, which leads to higher costs. In this case, the hash join is usually the better choice, as it does not require sorting and can handle large amounts of data well.

Nested loop here is inefficient, mainly because Publ is being used as the outer table. Even with an index on `Auth`, it remains significantly slower than merge and hash join.

Conclusion: a merge join with clustered indexes is the most efficient, but hash join is the best option without indexes.

Query 2

The second query contains a selection condition on `Auth.name`, which filters down the amount of tuples by several orders of magnitude. An indexed nested loop join in particular can benefit from this. This is recognized by the PostgreSQL planner as well.

If no suitable index is available, the hash join is once again a reasonably efficient option. The merge join with clustered indexes on the other hand is still clearly slower than the aforementioned two arrangements.

Conclusion: for selective conditions on a table, the nested loop with index on the countertable (here Publ) is clearly superior.

Time Spent on this Assignment

Time in hours per person: **4**

References

https://www.enterprisedb.com/postgres-tutorials/parallel-hash-joins-postgresql-explained?lang=en