

Assignment 4

Join Tuning

Database Tuning

Group A9

Fuetsch Sandro

Heigl David

May 26, 2026

Experimental Setup

All experiments were conducted on an MSI Titan laptop equipped with an NVMe SSD (sequential read up to 7 GB/s) and a multi-core CPU. The database system used was PostgreSQL (latest stable version) running locally on Windows. The dataset consists of the DBLP bibliography data with two tables:

- **Auth**(name, pubID) – 3 095 201 rows
- **Publ**(pubID, type, title, booktitle, year, publisher) – 1 233 214 rows

Data was loaded from tab-separated files (`publ.tsv` and `auth.tsv`). Each test configuration first reloads the data, then applies the specified index configuration, and finally runs the two queries using `EXPLAIN (ANALYZE, BUFFERS)`. Join strategies were forced using the PostgreSQL optimizer switches `enable_hashjoin`, `enable_mergejoin`, and `enable_nestloop`. Runtimes reported are execution times from `EXPLAIN ANALYZE`.

The two queries tested are:

Q1 (full join):

```
SELECT name, title FROM Auth, Publ
WHERE Auth.pubID = Publ.pubID;
```

Q2 (selective join):

```
SELECT title FROM Auth, Publ
WHERE Auth.pubID = Publ.pubID AND Auth.name = 'Divesh Srivastava';
```

1. Join Strategies Proposed by System

Response times

Indexes	Join Strategy Q1	Join Strategy Q2
no index	Hash Join, 8.42 s	Hash Join, 0.48 s
unique non-clustering on <code>Publ.pubID</code>	Nested Loop, 10.93 s	Nested Loop, 0.15 s
clustering on <code>Publ.pubID</code> and <code>Auth.pubID</code>	Merge Join, 10.29 s	Merge Join, 2.29 s

Discussion Without index: The optimizer chooses a *Hash Join*. This is expected because without any index, the only access path is a sequential scan on both tables. A hash join builds a hash table on the smaller relation (Publ, 1.2 M rows) and probes it with the larger relation (Auth, 3.1 M rows). This avoids the cost of sorting both tables (as required by merge join) and is generally the fastest strategy when no index is available. For Q2, the hash table is built on the 183 matching Auth rows (after filtering by name), making it very efficient (0.48 s).

With unique non-clustering index on Publ.pubID: The optimizer switches to a *Nested Loop Join* with an index scan on Publ. For Q1, Auth is scanned sequentially (outer relation, parallel), and for each of the 3.1 M Auth rows, the unique index on Publ is used to find the matching publication in $O(\log n)$ time. The total runtime is 10.93 s. For Q2, only 183 Auth rows match the name filter, so only 183 index lookups are needed, resulting in a very fast 0.15 s. The optimizer recognizes that a unique index guarantees at most one match per lookup, making nested loop very efficient – especially for selective queries.

With two clustering indexes: The optimizer chooses a *Merge Join*. Both tables are physically sorted by pubID (due to clustering), so the merge join can traverse both index scans in order without any additional sorting. For Q1, this yields 10.29 s. However, for Q2, the merge join is suboptimal (2.29 s vs. 0.15 s with nested loop): even though only 183 Auth rows match, the merge join still scans through the entire Publ index to find matches, because it must traverse the sorted order. The optimizer selects merge join because the clustering makes it appear cheap (no sort cost), but for highly selective queries, a nested loop with an index would be faster.

2. Indexed Nested Loop Join

Response times

Indexes	Response time Q1 [ms]	Response time Q2 [ms]
index on Publ.pubID	48 358	437
index on Auth.pubID	21 360	2 585
index on both	23 794	421

Query plans

Index on Publ.pubID (Q1):

```
Nested Loop  (cost=0.43..86811498.52 rows=3143511815 width=632)
    (actual time=0.072..47972.610 rows=3095201 loops=1)
    -> Seq Scan on auth  (rows=3095201 loops=1)
    -> Index Scan using publ_pubid_idx on publ
        Index Cond: (pubid = auth.pubid)
        Index Searches: 3095201
Execution Time: 48357.788 ms
```

Index on Publ.pubID (Q2):

```
Nested Loop  (cost=0.43..582547.48 rows=15717312 width=516)
    (actual time=26.051..437.418 rows=183 loops=1)
    -> Seq Scan on auth  Filter: (name = 'Divesh Srivastava')
        Rows Removed by Filter: 3095018
    -> Index Scan using publ_pubid_idx on publ
```

```
      Index Cond: (pubid = auth.pubid)
      Index Searches: 183
Execution Time: 437.485 ms
```

Index on Auth.pubID (Q1):

```
Nested Loop  (cost=0.43..525552888.42 rows=19085226030 width=632)
      (actual time=0.261..21066.685 rows=3095201 loops=1)
    -> Seq Scan on publ  (rows=1233214 loops=1)
    -> Index Scan using auth_pubid_idx on auth
          Index Cond: (pubid = publ.pubid)
          Index Searches: 1233214
Execution Time: 21359.878 ms
```

Index on Auth.pubID (Q2):

```
Gather  (actual time=304.572..2584.889 rows=183 loops=1)
  Workers Launched: 2
  -> Nested Loop
        -> Parallel Seq Scan on publ  (rows=411071.33 loops=3)
        -> Index Scan using auth_pubid_idx on auth
              Index Cond: (pubid = publ.pubid)
              Filter: (name = 'Divesh Srivastava')
              Rows Removed by Filter: 3
              Index Searches: 1233214
Execution Time: 2584.937 ms
```

Index on both (Q1):

```
Nested Loop  (cost=0.43..525552888.42 rows=19085226030 width=632)
      (actual time=0.237..23463.255 rows=3095201 loops=1)
    -> Seq Scan on publ  (rows=1233214 loops=1)
    -> Index Scan using auth_pubid_idx on auth
          Index Cond: (pubid = publ.pubid)
          Index Searches: 1233214
Execution Time: 23793.559 ms
```

Index on both (Q2):

```
Nested Loop  (actual time=42.161..420.501 rows=183 loops=1)
    -> Seq Scan on auth  Filter: (name = 'Divesh Srivastava')
          Rows Removed by Filter: 3095018
    -> Index Scan using publ_pubid_idx on publ
          Index Cond: (pubid = auth.pubid)
          Index Searches: 183
Execution Time: 420.558 ms
```

Discussion Index on Publ.pubID only: For Q1, Auth is the outer relation (3.1M rows scanned sequentially), and for each row, the index on Publ is probed. This results in 3.1M index lookups, which is very expensive (48.4s). For Q2, the name filter reduces Auth to 183 rows, so only 183 index lookups are needed – hence the fast 437 ms.

Index on Auth.pubID only: For Q1, the optimizer reverses the join order: Publ becomes the outer relation (1.2M rows), and for each Publ row, the Auth index is probed. Since Publ is smaller, only 1.2 M lookups are needed (vs. 3.1M), yielding 21.4s

– roughly half the time. However, for Q2, this is much slower (2.6 s): the optimizer scans all of Publ (1.2 M rows) and for each row looks up Auth and then filters by name. The selectivity filter on **Auth.name** cannot be applied early because there is no index on name; the filter happens after the index lookup on Auth.pubID, meaning all 1.2 M lookups still occur.

Indexes on both: For Q1, the optimizer uses Auth.pubID as the inner index (same as Auth-only case), resulting in similar performance (23.8 s). For Q2, the optimizer now chooses the better plan: scan Auth with the name filter first (183 rows), then use the Publ index for lookups (183 lookups), yielding 421 ms – comparable to the Publ-only case. Having both indexes gives the optimizer the freedom to pick the optimal join direction for each query.

The key insight is that the number of index lookups determines performance. The ideal configuration uses the more selective side as the outer relation and probes the indexed inner relation. For Q2, filtering Auth by name first and then looking up Publ is always optimal, regardless of which indexes exist.

3. Sort-Merge Join

Response times

Indexes	Response time Q1 [ms]	Response time Q2 [ms]
no index	46 599	14 635
two non-clustering indexes	10 198	2 881
two clustering indexes	8 038	2 151

Query plans

No index (Q1):

```
Merge Join (actual time=36241.334..46182.118 rows=3095201 loops=1)
  Merge Cond: (publ.pubid = auth.pubid)
    -> Sort Sort Key: publ.pubid
      Sort Method: external merge Disk: 121600kB
      -> Seq Scan on publ (rows=1233214)
    -> Materialize
      -> Sort Sort Key: auth.pubid
        Sort Method: external merge Disk: 148328kB
        -> Seq Scan on auth (rows=3095201)
Execution Time: 46599.498 ms
```

No index (Q2):

```
Merge Join (actual time=12529.969..14608.658 rows=183 loops=1)
  Merge Cond: (auth.pubid = publ.pubid)
    -> Sort Sort Key: auth.pubid
      Sort Method: quicksort Memory: 25kB
      -> Gather -> Parallel Seq Scan on auth
        Filter: (name = 'Divesh Srivastava')
    -> Materialize
      -> Sort Sort Key: publ.pubid
        Sort Method: external merge Disk: 121600kB
        -> Seq Scan on publ (rows=1233214)
Execution Time: 14634.974 ms
```

Two non-clustering indexes (Q1):

```
Merge Join (actual time=0.102..9923.931 rows=3095201 loops=1)
  Merge Cond: (publ.pubid = auth.pubid)
    -> Index Scan using publ_pubid_idx on publ
      Index Searches: 1
    -> Materialize
      -> Index Scan using auth_pubid_idx on auth
        Index Searches: 1
Execution Time: 10198.036 ms
```

Two non-clustering indexes (Q2):

```
Merge Join (actual time=327.404..2881.092 rows=183 loops=1)
  Merge Cond: (publ.pubid = auth.pubid)
    -> Index Scan using publ_pubid_idx on publ (rows=1229958)
      Index Searches: 1
    -> Sort Sort Key: auth.pubid
      Sort Method: quicksort Memory: 25kB
      -> Gather -> Parallel Seq Scan on auth
        Filter: (name = 'Divesh Srivastava')
Execution Time: 2881.173 ms
```

Two clustering indexes (Q1):

```
Merge Join (actual time=0.122..7781.063 rows=3095201 loops=1)
  Merge Cond: (publ.pubid = auth.pubid)
    -> Index Scan using publ_pubid_idx on publ
      Index Searches: 1
    -> Materialize
      -> Index Scan using auth_pubid_idx on auth
        Index Searches: 1
Execution Time: 8038.208 ms
```

Two clustering indexes (Q2):

```
Merge Join (actual time=280.926..2150.606 rows=183 loops=1)
  Merge Cond: (publ.pubid = auth.pubid)
    -> Index Scan using publ_pubid_idx on publ (rows=1229958)
    -> Sort Sort Key: auth.pubid
      Sort Method: quicksort Memory: 25kB
      -> Gather -> Parallel Seq Scan on auth
        Filter: (name = 'Divesh Srivastava')
Execution Time: 2150.691 ms
```

Discussion No index: Both tables must be fully sorted before the merge can begin. The sort uses *external merge* (spilling to disk) because the tables are too large for `work_mem`. Publ requires 121 MB and Auth requires 148 MB of disk for sorting. This dominates the runtime: Q1 takes 46.6 s, of which the majority is spent sorting. For Q2, the Auth side is much smaller after filtering (183 rows, sorted in memory with quicksort), but Publ still needs a full external sort (121 MB), resulting in 14.6 s.

Two non-clustering indexes: The indexes provide a sorted access path, eliminating the need for explicit sorting. The merge join simply traverses both index scans in order. Q1 drops from 46.6 s to 10.2 s – a $4.5\times$ improvement. However, since the indexes are

non-clustering, the actual data pages are not physically sorted, leading to random I/O when fetching the tuple data. For Q2, the improvement is less dramatic (14.6 s to 2.9 s) because the Publ index scan must still traverse much of the index.

Two clustering indexes: With clustering, the physical order of the table matches the index order. This means sequential I/O instead of random I/O during the index scan. Q1 improves further to 8.0 s (vs. 10.2 s non-clustering), a 22% improvement due to better cache utilization and sequential read patterns. For Q2, the improvement is modest (2.9 s to 2.2 s). The bottleneck for Q2 remains scanning through the Publ index to find the 183 matching pubIDs; clustering helps with I/O patterns but does not reduce the number of comparisons.

The merge join is well-suited for Q1 (joining all rows) because both sides are similar in size and the sorted traversal is efficient. For Q2, where only 183 rows from Auth match, the merge join wastes effort scanning through Publ – a nested loop with an index (0.15 s) would be far superior.

4. Hash Join

Response times

Indexes	Response time Q1 [ms]	Response time Q2 [ms]
no index	8 424	477

Query plans

No Index (Q1):

```
Hash Join (actual time=1172.732..8125.193 rows=3095201 loops=1)
  Hash Cond: (auth.pubid = publ.pubid)
    -> Seq Scan on auth (rows=3095201)
    -> Hash Buckets: 16384 Batches: 32 (originally 16)
        Memory Usage: 8065kB
    -> Seq Scan on publ (rows=1233214)
Execution Time: 8424.266 ms
```

No Index (Q2):

```
Hash Join (actual time=140.546..476.948 rows=183 loops=1)
  Hash Cond: (publ.pubid = auth.pubid)
    -> Seq Scan on publ (rows=1233214)
    -> Hash Buckets: 4096 Batches: 1 Memory Usage: 43kB
        -> Gather -> Parallel Seq Scan on auth
            Filter: (name = 'Divesh Srivastava')
Execution Time: 477.007 ms
```

Discussion The hash join performs remarkably well *without any index*:

Q1 (8.4 s): This is the fastest result for the full join among all strategies tested without indexes. Compared to sort-merge without index (46.6 s), the hash join is $5.5\times$ faster. The reason is that hash join avoids the expensive external sort: it builds a hash table on Publ (the smaller relation) in 1.2 s, then probes it with each Auth row. Each probe is $O(1)$ amortized. The hash table required 32 batches (spilling to temp files), indicating it did not fit entirely in `work_mem`, but even with spilling, hashing is cheaper than sorting.

Compared to the indexed nested loop (best case 21.4s with index on Auth.pubID), the hash join is still $2.5\times$ faster. The nested loop performs 1.2M B-tree lookups (each $O(\log n)$), which is more expensive than 3.1M hash probes (each $O(1)$).

Even compared to sort-merge with clustering indexes (8.0s), the hash join without any index is competitive (8.4s). This demonstrates that for equi-joins on large tables, hash join is often the best strategy.

Q2 (477 ms): For the selective query, the hash join builds a tiny hash table on the 183 filtered Auth rows (only 43 kB, fits entirely in memory with 1 batch), then probes it while scanning Publ. This is fast (477ms) but not the fastest: the indexed nested loop with an index on Publ.pubID achieves 150ms by performing only 183 index lookups instead of scanning all of Publ. The hash join must still perform a full sequential scan of Publ (1.2M rows), which is its main cost for selective queries.

Summary: Hash join is the best all-round strategy for equi-joins, especially for large, unindexed tables. Its weakness appears only in highly selective queries where an indexed nested loop can exploit the selectivity to avoid scanning the larger table entirely.

Time Spent on this Assignment

Time in hours per person: **6**

References

1. PostgreSQL Documentation: Query Planning, <https://www.postgresql.org/docs/current/runtime>
2. PostgreSQL Documentation: EXPLAIN, <https://www.postgresql.org/docs/current/sql-explain>
3. PostgreSQL Documentation: CREATE INDEX / CLUSTER, <https://www.postgresql.org/docs/c>
4. H. Garcia-Molina, J. D. Ullman, J. Widom: *Database Systems: The Complete Book*, 2nd Edition, Pearson, 2008. Chapters on Join Algorithms.