**Database Tuning – Assignment 5**

# Join Tuning

A2

Baumgartner Dominik, 0920177

Dafir Thomas Samy, 1331483

Schörgnhofer Kevin, 1421082

December 13, 2016

## 1 Setup

All queries were sent to the database-server (biber) using *psql* on the computers of the RÜR.

## 2 Join Strategies Proposed by System

**Response times**

| Indexes | Join Strategy Q1 | Join Strategy Q2 |
|---|---|---|
| no index | Hash Join | Hash Join |
| unique non-clustering on `Publ.pubID` | Hash Join | Nested Loop Join |
| clustering on `Publ.pubID` and `Auth.pubID` | Merge Join | Nested Loop Join |

**Discussion** Discuss here your observations. Is the choice of the strategy expected? How does the system come to this choice?
no index:
For this scenario the proposed join strategies are the expected ones. A hash join is in this case the best choice. For a merge join the tables should be sorted and the nested loop join should be used when there is a small table.

unique non-clustering on `Publ.pubID`:
For the first query a hash join is the best choice, because of the big, unsorted table of both. For this query the index is of no use.
For the second query a Nested Loop Join is used because first the name from the author must be searched which reduces the table size from $3 * 10^6$ to ∼200. Thereforce the nested loop join is the better choice.

clustering on `Publ.pubID` and `Auth.pubID`:
For the first query a Merge Join is proposed as expected. The reason therefore is that both tables are sorted to this attribute which is ideal for the Merge Join.
For the second query we expected a Merge Join, but the system used a Nested Loop Join. The Reason therefore is again the table size. First the name is searched in Auth

which reduces the size and also ”destroys” the ordering. This means that the system would have to sort the values again for the Merge Join although the values are still sorted. And therefore a Nested Loop Join is the best choice.

## 3 Nested Loop Join

### Response times

| Indexes | Response time Q1 [ms] | Response time Q2 [ms] |
|---|---:|---:|
| index on `Publ.pubID` | 98635 | 504 |
| index on `Auth.pubID` | 46672 | 193800 |
| index on `Publ.pubID` and `Auth.pubID` | 56395 | 511 |

### Query plans

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

```
Q1:
 Nested Loop  (cost=0.43..1589599.47 rows=3095201 width=82)
 (actual time=0.088..97373.109 rows=3095201 loops=1)
   ->  Seq Scan on auth  (cost=0.00..57761.01 rows=3095201 width=38)
       (actual time=0.012..1694.669 rows=3095201 loops=1)
   ->  Index Scan using pubidp on publ_i  (cost=0.43..0.48 rows=1 width=89)
       (actual time=0.028..0.029 rows=1 loops=3095201)
         Index Cond: ((pubid)::text = (auth.pubid)::text)
 Planning time: 0.647 ms
 Execution time: 98634.857 ms

Q2:
 Nested Loop  (cost=0.43..65701.99 rows=24 width=67)
 (actual time=309.784..503.737 rows=183 loops=1)
   ->  Seq Scan on auth  (cost=0.00..65499.01 rows=24 width=23)
       (actual time=309.720..498.274 rows=183 loops=1)
         Filter: ((name)::text = 'Divesh Srivastava'::text)
         Rows Removed by Filter: 3095018
   ->  Index Scan using pubidp on publ_i  (cost=0.43..8.45 rows=1 width=89)
       (actual time=0.027..0.028 rows=1 loops=183)
         Index Cond: ((pubid)::text = (auth.pubid)::text)
 Planning time: 0.137 ms
 Execution time: 503.837 ms
```

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

```
Q1:
 Nested Loop  (cost=0.43..849985.42 rows=3095201 width=82)
 (actual time=0.066..45483.489 rows=3095201 loops=1)
   ->  Seq Scan on publ  (cost=0.00..34694.14 rows=1233214 width=89)
       (actual time=0.009..703.605 rows=1233214 loops=1)
   ->  Index Scan using pubida on auth_i  (cost=0.43..0.63 rows=3 width=38)
       (actual time=0.026..0.033 rows=3 loops=1233214)
         Index Cond: ((pubid)::text = (publ.pubid)::text)
 Planning time: 0.432 ms
 Execution time: 46672.318 ms
```

```
Q2:
 Nested Loop  (cost=0.00..562648.46 rows=25 width=67)
 (actual time=15405.220..193800.011 rows=183 loops=1)
   Join Filter: ((auth_i.pubid)::text = (publ.pubid)::text)
   Rows Removed by Join Filter: 225677979
   ->  Seq Scan on publ  (cost=0.00..34694.14 rows=1233214 width=89)
       (actual time=0.012..689.233 rows=1233214 loops=1)
   ->  Materialize  (cost=0.00..65499.14 rows=25 width=23)
       (actual time=0.000..0.071 rows=183 loops=1233214)
         ->  Seq Scan on auth_i  (cost=0.00..65499.01 rows=25 width=23)
             (actual time=104.495..507.970 rows=183 loops=1)
               Filter: ((name)::text = 'Divesh Srivastava'::text)
               Rows Removed by Filter: 3095018
 Planning time: 0.135 ms
 Execution time: 193800.146 ms
```

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

```
Q1:
 Nested Loop  (cost=0.43..850049.42 rows=3095201 width=82)
 (actual time=0.101..55181.800 rows=3095201 loops=1)
   ->  Seq Scan on publ_i  (cost=0.00..34758.14 rows=1233214 width=89)
       (actual time=0.037..721.909 rows=1233214 loops=1)
   ->  Index Scan using pubida on auth_i  (cost=0.43..0.63 rows=3 width=38)
       (actual time=0.032..0.041 rows=3 loops=1233214)
         Index Cond: ((pubid)::text = (publ_i.pubid)::text)
 Planning time: 0.148 ms
 Execution time: 56394.801 ms

Q2:
 Nested Loop  (cost=0.43..65710.45 rows=25 width=67)
 (actual time=103.244..511.228 rows=183 loops=1)
   ->  Seq Scan on auth_i  (cost=0.00..65499.01 rows=25 width=23)
       (actual time=103.176..505.597 rows=183 loops=1)
         Filter: ((name)::text = 'Divesh Srivastava'::text)
         Rows Removed by Filter: 3095018
   ->  Index Scan using pubidp on publ_i  (cost=0.43..8.45 rows=1 width=89)
       (actual time=0.028..0.029 rows=1 loops=183)
         Index Cond: ((pubid)::text = (auth_i.pubid)::text)
 Planning time: 0.181 ms
 Execution time: 511.343 ms
```

**Discussion** Index on Publ.pubID - Q1:
As expected publ is the inner relation, because we have an index on the join attribute on publ, which saves access time for the DBS. This means the DBS can use a index scan on publ, but has to use a seq scan on auth. This is as expected (for the nested loop join) the slowest approach for Q1, because the index is on the smaller of the two relations.
Index on Publ.pubID - Q2:
Again as in Q1 the DBS can use Publ.pubID as index and uses publ as inner relation, the only difference is that we filter auth with the selection on name, which results in a way smaller relation for the join. This explains the vast difference between the runtimes of Q1 and Q2.
Index on Auth.pubID - Q1:
In this case the DBS chooses auth as inner relation, because it can access it via the

index. In fact the order of the relations is just the other way around compared to 'Index on Publ.pubID - Q1'. Of course the runtime is in this case smaller, because this time we have the index on the larger relation.
Index on Auth.pubID - Q2:
At a first glance the runtime for this one seems pretty high. The problem here is that the DBS can't benefit from the index, because it has to filter auth by names. So the only option in this case is a seq scan on auth. After the filtering, auth is again smaller than publ, which is the reason why auth is the inner relation. Given that we have no index on publ we have to go here for a seq scan again. This explains the relatively enormous runtime.
Index on Publ.pubID and Auth.pubID - Q1:
The query plan shows that auth is used as inner relation, which is as expected, because both relations have indexes, but auth is the one with more tuples. Also as expected is, that the outer query gets accessed via seq scan (we have to check every tuple anyway, so the index would be more hindering than helpful). This ends up in a query plan which is (basically) the same as the one from 'Index on Auth.pubID - Q1', which is not astonishing. The only curious fact here is the high difference in the runtime, which we have no explanation for. Index on Publ.pubID and Auth.pubID - Q2:
As well as the previous query, this query has a "twin-query" too. It is basically the same as 'Index on Publ.pubID - Q2'. We have to access auth via seq scan because we have to apply the name filter (were we can't use the index on pubID). Since we have an index on publ, we use publ as the inner relation, hence auth as outer relation. Again (similar to our previous twins) the runtime is not exactly the same, but this time much more closer.

## 4 Sort-Merge Join

### Response times

| Indexes | Response time Q1 [ms] | Response time Q2 [ms] |
|---|---:|---:|
| no index | too long | 27886 |
| two non-clustering indexes | 37306 | 508 |
| two clustering indexes | 18891 | 515 |

### Query plans

No index (Q1/Q2):

```
Q1:
Merge Join  (cost=846625.43..906956.29 rows=3095201 width=83)
   Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
   ->  Sort  (cost=285913.35..288996.38 rows=1233214 width=90)
         Sort Key: publ.pubid
         ->  Seq Scan on publ  (cost=0.00..34694.14 rows=1233214 width=90)
   ->  Materialize  (cost=560711.47..576187.47 rows=3095201 width=38)
         ->  Sort  (cost=560711.47..568449.47 rows=3095201 width=38)
               Sort Key: auth.pubid
               ->  Seq Scan on auth  (cost=0.00..57750.01 rows=3095201 width=38)

Q2:
Merge Join  (cost=351419.92..357590.96 rows=413 width=68)
    (actual time=24054.639..27857.060 rows=183 loops=1)
   Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
   ->  Sort  (cost=285913.35..288996.38 rows=1233214 width=90)
             (actual time=23183.951..26169.586 rows=1229958 loops=1)
         Sort Key: publ.pubid
         Sort Method: external merge  Disk: 121400kB
         ->  Seq Scan on publ  (cost=0.00..34694.14 rows=1233214 width=90)
                               (actual time=0.026..850.908 rows=1233214 loops=1)
   ->  Sort  (cost=65505.96..65506.99 rows=413 width=23)
     (actual time=519.464..519.550 rows=183 loops=1)
         Sort Key: auth.pubid
         Sort Method: quicksort  Memory: 39kB
         ->  Seq Scan on auth  (cost=0.00..65488.01 rows=413 width=23)
                               (actual time=8.420..518.464 rows=183 loops=1)
               Filter: ((name)::text = 'Divesh Srivastava'::text)
               Rows Removed by Filter: 3095018
 Planning time: 0.351 ms
 Execution time: 27886.590 ms
```

Naturally, if there are no indexes present on the merge-attributes, both relations have to be sorted on the merge-attribute. This takes place in the execution of both queries. In Query 1 we see a sorting of both relations followed by a merge-join on the specified attribute (The Query took extremely long, so the execution was stopped). query 2 is much faster since first a selection is consucted using a linear scan since there is no index on name. After that both relations are again sorted which is a lot faster since the second only contains publications by one author. Just like in query 1 the system then uses a merge-join to get the result.

Two non-clustering indexes (Q1/Q2):

```
Q1:
Merge Join  (cost=0.86..263625.34 rows=3095201 width=83)
            (actual time=0.038..36048.698 rows=3095201 loops=1)
   Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
   ->  Index Scan using pubidpubl on publ  (cost=0.43..73055.43 rows=1233214 width=90)
   (actual time=0.006..6608.030 rows=1233208 loops=1)
   ->  Index Scan using pubidauth on auth  (cost=0.43..148974.61 rows=3095201 width=38)
           (actual time=0.005..13439.998 rows=3095201 loops=1)
 Planning time: 0.768 ms
 Execution time: 37306.638 ms

Q2:
Merge Join  (cost=65499.99..141460.64 rows=24 width=68)
    (actual time=508.011..508.011 rows=0 loops=1)
   Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
   ->  Index Scan using pubidpubl on publ  (cost=0.43..73055.43 rows=1233214 width=90)
   (actual time=0.006..0.006 rows=1 loops=1)
   ->  Sort  (cost=65499.56..65499.62 rows=24 width=23)
     (actual time=508.002..508.002 rows=0 loops=1)
         Sort Key: auth.pubid
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on auth  (cost=0.00..65499.01 rows=24 width=23)
       (actual time=507.995..507.995 rows=0 loops=1)
               Filter: ((name)::text = 'Divesh Srivastav'::text)
               Rows Removed by Filter: 3095201
 Planning time: 0.528 ms
 Execution time: 508.045 ms
```

In this case although we do not have physical sorting of the data itself, we can use the index (always sorted) to access the tables. This eliminates the need for sorting. In query 1 we see that sorting has completely been removed. The system uses both non-clustering indexes to conduct a merge-join. This of course speeds up the query dramatically. In query 2 we still see some sorting which is only due to the fact that the result eturned by the sequential scan (selection of name) can not be assumed to be sorted. However this sort is only carried out on a small amount of tuples. afterwards the merge-join is executed using the index of the first elation and the sorted second relation. This method is still not optimal since only the index is sorted whereas the actual tupes are still spread out across the disk which means that tuples with the similar attribute-values are not necessaily close to each other. This prohibits the system from performing sequential reads which slows down query execution compared to a clustering index.

Two clustering indexes (Q1/Q2):

```
Q1:
Merge Join  (cost=0.86..263629.21 rows=3095201 width=83)
    (actual time=0.022..17687.042 rows=3095201 loops=1)
   Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
   ->  Index Scan using pubidpubl on publ  (cost=0.43..73057.97 rows=1233214 width=90)
   (actual time=0.005..917.866 rows=1233208 loops=1)
   ->  Index Scan using pubidauth on auth  (cost=0.43..148975.94 rows=3095201 width=38)
   (actual time=0.004..2033.002 rows=3095201 loops=1)
 Planning time: 0.606 ms
 Execution time: 18891.404 ms

Q2:
Merge Join  (cost=65500.99..141464.18 rows=24 width=68)
    (actual time=515.935..515.935 rows=0 loops=1)
   Merge Cond: ((publ.pubid)::text = (auth.pubid)::text)
   ->  Index Scan using pubidpubl on publ  (cost=0.43..73057.97 rows=1233214 width=90)
                (actual time=0.006..0.006 rows=1 loops=1)
   ->  Sort  (cost=65500.56..65500.62 rows=24 width=23)
     (actual time=515.926..515.926 rows=0 loops=1)
         Sort Key: auth.pubid
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on auth  (cost=0.00..65500.01 rows=24 width=23)
                               (actual time=515.919..515.919 rows=0 loops=1)
               Filter: ((name)::text = 'Divesh Srivastav'::text)
               Rows Removed by Filter: 3095201
 Planning time: 0.535 ms
 Execution time: 515.970 ms
```

The increase in execution-speed cannot be explained by looking at the query plans alone since the look exactly the same as the ones in the non-clustered example. However the situation is a little different. Since there are clusteringindexes on both relations they are of course physically sorted on the disk. Execution of both queries is exactly the same as in the previous example. The queries aresped up since the physical sorted tuples can now be sequentially read from the disk, which gives a pretty significant speed boost as can be notced by comparing the execution durations.

**Discussion** The observations as already discussed for each query plan were all as expected. Using a merge join when there are no indexes present is of course extremely slow due to both relations having to be physically sorted. A non-clustering index speeds things up since no more sorting is required. Tuples can be accessed through the index however sequential disk-read is not possible. Finally using two clustered indexes eliminates the need for sorting and sequential disk-reads can be used. Looking at the first query we see huge gains in execution time due to the factors previously explained. The second query is also hugely sped up by the use of indexes vs no index. However there is not a large difference in execution time between using clustering and non-clustering indexes. Since this is not the case in query 1 is has to be caused by the selection. After the selection the system can no longer use the index on the second relation and has to sort the result of the selection. This part is the same in both cases (clustering/non-clustering). So it is likely that the sequential scan and following sort dominate the execution time and the differende between seuqential- and random-read of the first relation does not have a big impact.

## 5 Hash Join

### Response times

| Indexes | Response time Q1 [ms] | Response time [ms] Q2 |
|---|---:|---:|
| no index | 9427 | 1668 |

### Query plans

No Index (Q1/Q2):

```
Q1:
 Hash Join  (cost=68174.32..250399.34 rows=3095201 width=82)
 (actual time=1834.128..8248.005 rows=3095201 loops=1)
   Hash Cond: ((auth.pubid)::text = (publ.pubid)::text)
   ->  Seq Scan on auth  (cost=0.00..57761.01 rows=3095201 width=38)
       (actual time=0.011..1413.767 rows=3095201 loops=1)
   ->  Hash  (cost=34694.14..34694.14 rows=1233214 width=89)
       (actual time=1833.830..1833.830 rows=1233214 loops=1)
         Buckets: 4096  Batches: 64  Memory Usage: 2344kB
         ->  Seq Scan on publ  (cost=0.00..34694.14 rows=1233214 width=89)
             (actual time=0.042..790.847 rows=1233214 loops=1)
 Planning time: 0.445 ms
 Execution time: 9426.895 ms

Q2:
 Hash Join  (cost=65499.31..140273.15 rows=24 width=67)
 (actual time=607.564..1667.832 rows=183 loops=1)
   Hash Cond: ((publ.pubid)::text = (auth.pubid)::text)
   ->  Seq Scan on publ  (cost=0.00..34694.14 rows=1233214 width=89)
       (actual time=0.045..592.658 rows=1233214 loops=1)
   ->  Hash  (cost=65499.01..65499.01 rows=24 width=23)
       (actual time=516.867..516.867 rows=183 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 11kB
         ->  Seq Scan on auth  (cost=0.00..65499.01 rows=24 width=23)
             (actual time=321.786..516.733 rows=183 loops=1)
               Filter: ((name)::text = 'Divesh Srivastava'::text)
               Rows Removed by Filter: 3095018
 Planning time: 0.111 ms
 Execution time: 1667.943 ms
```

**Discussion** About the query plans: Q1 uses publ as build input as expected, because it is the relation with less tuple. In the case of Q2 the DBS uses auth as build input, because after the selection (name = 'Divesh Srivastava') the rest of auth is smaller than publ.

Response time: Compared to the merge-join is faster for query 1 which is expected for no index and non-clustering indexex since. If there are no indexes both relations have to be sorted which is expensive. In case of two non-clsutering indexes data can be accessed through the always sorted index, but these data-accesses are random since the suples are not sorted on the disk which leads to many blocks being read more than once and explaines why the hash join is faster. The fact that a merge join using two clustering indexex is slower than a hssh join is not expected since in theory a merge-join on of two sorted relations should require only about 1/3 of block accesses a hash join requires. For query 2 all results were expected. The no-index-case is extremely slow. both other cases are faster than the hash join since the second relation is really small and a hash join has a big overhead. Looking at query 1 the hash join is a lot faster than the nested loop join no matter which indexes are used. This is as expected since in this case nested loop joins are index-nested loop joins which have to do an index lookup for each tuple in the outer relation. Both relations are quite large, so $3 * (b_r + b_s)$ is a lot smaller than $b_r + n_r * c$ where c are the block accesses for an index lookup.

Time in hours per person: **XXX**

**Important:** Reference your information sources!