Assignment 4

Index Tuning – Selection

Database Tuning

New Group 8

Frauenschuh Florian, 12109584

Lindner Peter, 12101607

Weilert Alexander, 12119653

May 27, 2024

Notes

- Do not forget to run `ANALYZE tablename` after creating or changing a table.
- Use `EXPLAIN ANALYZE` for the query plans that you display in the report.

Experimental Setup

How do you send the queries to the database? How do you measure the execution time for a sequence of queries?

For our experiments we used the following hardware and software:

Component	Specs
Processor	i7-13700H 3.7-5.0 GHz
Memory	32 GiB

Table 1: Hardware: Dell XPS 15 9530

Software	Version
OS	Ubuntu 22.04
Postgres	2.3.4
postgresql	42.7.3
MariaDB	10.6.16
mariadb-java-client	3.3.3
Java	18

Table 2: Software

Postgres was hosted on localhost, on which we also executed our experiments. The client was implemented in Java and gained access to our databases using the JDBC drivers listed above.

For the queries we used prepared statements, and the performance time was measured by determining the throughput of the queries in a 60-second time frame.

Clustering B^+ Tree Index

Point Query Repeat the following query multiple times with different conditions for pubID.

```
SELECT * FROM Publ WHERE pubID = ...
```

Which conditions did you use?

For each query a random existing pubID was chosen.

Show the runtime results and compute the throughput.

Executed Queries: 1985347
Queries per Second: 33089.12

Query plan (for one of the queries):

```
Index Scan using idx_clustering_pubid on publ  (cost=0.43..8.45 rows=1 width=112)
                                                (actual time=0.051..0.052 rows=1 loops=1)
  Index Cond: ((pubid)::text = 'books/acm/kim95/Kim95'::text)
Planning Time: 0.189 ms
Execution Time: 0.067 ms
```

An index scan on the pubID is performed, which make use of the B^+ tree index.

Multipoint Query vs. Multipoint Query IN-Predicate – Low Selectivity Repeat the following query multiple times with different conditions for booktitle.

```
SELECT * FROM Publ WHERE booktitle = ...
```

```
SELECT * FROM Publ WHERE pubID IN (...)
```

Which conditions did you use?

To maintain low selectivity, we are using random non-empty book titles out of the dataset. Similarly, for the authors, we are choosing three random authors, but do not need to specify the "non-empty" condition, since an author is always given.

Show the runtime results and compute the throughput.

Multipoint Query
Executed Queries: 146054
Queries per Second: 2434.23

Multipoint Query IN-Predicate
Executed Queries: 82317
Queries per Second: 1371.95

Query plan (for one of the queries):

Multipoint Query

```
Index Scan using idx_clustering_booktitle on publ  (cost=0.43..14.56 rows=179 width=113)
                                                    (actual time=0.021..0.025 rows=34 loops=1)
  Index Cond: ((booktitle)::text = 'Modern Database Systems'::text)
Planning Time: 0.058 ms
Execution Time: 0.035 ms
```

Again, the query plan shows an B^+ tree index scan on the booktitle.

Multipoint Query IN-Predicate

```
Nested Loop  (cost=19.16..628.21 rows=72 width=112)
             (actual time=0.200..4.579 rows=294 loops=1)
  ->  HashAggregate  (cost=18.73..19.45 rows=72 width=23)
                     (actual time=0.171..0.229 rows=294 loops=1)
        Group Key: (auth.pubid)::text
        Batches: 1  Memory Usage: 77kB
        ->  Index Scan using idx_clustering_name on auth
                              (cost=0.43..18.55 rows=72 width=23)
                              (actual time=0.041..0.114 rows=294 loops=1)
"              Index Cond: ((name)::text = ANY
                           (’{""William Kent"",""Alfons Kemper"",""Walid G. Aref""}’::text[]))"
  ->  Index Scan using idx_clustering_pubid on publ
                           (cost=0.43..8.45 rows=1 width=112)
                           (actual time=0.014..0.014 rows=1 loops=294)
        Index Cond: ((pubid)::text = (auth.pubid)::text)
Planning Time: 0.824 ms
Execution Time: 4.625 ms
```

This query plan is significantly more complex, as it involves a nested loop and a hash aggregate. First, an index scan is performed on the `name` attribute of the `auth` table, followed by a hash aggregate operation, due to the use of the `IN` predicate. Then, an index scan is performed on the `pubID` attribute of the `publ` table. Finally, the results are joined using a nested loop.

Multipoint Query – High Selectivity Repeat the following query multiple times with different conditions for `year`.

```
SELECT * FROM Publ WHERE year = ...
```

Which conditions did you use?

Same as before, we are using random years from the dataset.

Show the runtime results and compute the throughput.

Executed Queries: 2463
Queries per Second: 41.05

Query plan (for one of the queries):

```
Index Scan using idx_clustering_year on publ  (cost=0.43..2068.83 rows=52823 width=113)
                                              (actual time=0.051..6.806 rows=50803 loops=1)
  Index Cond: ((year)::text = ’2000’::text)
Planning Time: 0.054 ms
Execution Time: 7.923 ms
```

Similar to the previous query plans, an index scan is performed on the `year` attribute.

Non-Clustering B^+ Tree Index

Note: Make sure the data is not physically ordered by the indexed attributes due to the clustering index that you created before.

Point Query Repeat the following query multiple times with different conditions for `pubID`.

```
SELECT * FROM Publ WHERE pubID = ...
```

Which conditions did you use?

For each query a random existing pubID was chosen.

Show the runtime results and compute the throughput.

Executed Queries: 1973107
Queries per Second: 32885.11

Query plan (for one of the queries):

```
Index Scan using idx_clustering_pubid on publ  (cost=0.43..8.45 rows=1 width=112)
                                               (actual time=0.024..0.024 rows=1 loops=1)
  Index Cond: ((pubid)::text = 'books/acm/kim95/Kim95'::text)
Planning Time: 0.052 ms
Execution Time: 0.034 ms
```

The query makes use of the B^+ tree index on the pubID attribute, just as with the clustering index.

Multipoint Query vs. Multipoint Query IN-Predicate – Low Selectivity Repeat the following query multiple times with different conditions for booktitle.

```
SELECT * FROM Publ WHERE booktitle = ...
```

```
SELECT * FROM Publ WHERE pubID IN (...)
```

Which conditions did you use?

To maintain low selectivity, we are using random non-empty book titles out of the dataset. Similarly, for the authors, we are choosing three random authors, but do not need to specify the "non-empty" condition, since an author is always given.

Show the runtime results and compute the throughput.

Multipoint Query
Executed Queries: 147930
Queries per Second: 2465.5

Multipoint Query IN-Predicate
Executed Queries: 80356
Queries per Second: 1339.27

Query plan (for one of the queries):

Multipoint Query

```
Index Scan using idx_clustering_booktitle on publ  (cost=0.43..661.90 rows=179 width=112)
                                                    (actual time=0.049..0.051 rows=34 loops=1)
  Index Cond: ((booktitle)::text = 'Modern Database Systems'::text)
Planning Time: 0.172 ms
Execution Time: 0.061 ms
```

Again an index scan is performed on the booktitle attribute of the publ table.

Multipoint Query IN-Predicate

```
Nested Loop  (cost=19.16..628.21 rows=72 width=112)
             (actual time=0.129..2.430 rows=294 loops=1)
  ->  HashAggregate  (cost=18.73..19.45 rows=72 width=23)
                     (actual time=0.119..0.163 rows=294 loops=1)
        Group Key: (auth.pubid)::text
        Batches: 1  Memory Usage: 77kB
        ->  Index Scan using idx_clustering_name on auth  (cost=0.43..18.55 rows=72 width=23)
                                                          (actual time=0.014..0.062 rows=294 loops=1)"
                Index Cond: ((name)::text = ANY
                            ('{""William Kent"",""Alfons Kemper"",""Walid G. Aref""}'::text[]))"
  ->  Index Scan using idx_clustering_pubid on publ  (cost=0.43..8.45 rows=1 width=112)
                                                     (actual time=0.007..0.007 rows=1 loops=294)
        Index Cond: ((pubid)::text = (auth.pubid)::text)
Planning Time: 0.204 ms
Execution Time: 2.464 ms
```

First, an index scan with the three chosen authors as in the condition is performed on the name index of the auth table. Then, similarly to the clustering index, the hash aggregate operation is performed, followed by an index scan on the pubID attribute of the publ table. Finally, the results are joined using a nested loop.

Multipoint Query – High Selectivity Repeat the following query multiple times with different conditions for year.

```
SELECT * FROM Publ WHERE year = ...
```

Which conditions did you use?

Same as before, we are using random years from the dataset.

Show the runtime results and compute the throughput.

Executed Queries: 2482
Queries per Second: 41.37

Query plan (for one of the queries):

```
Bitmap Heap Scan on publ  (cost=537.16..23565.35 rows=48095 width=112)
                            (actual time=2.858..51.485 rows=50803 loops=1)
  Recheck Cond: ((year)::text = '2000'::text)
  Heap Blocks: exact=11570
  ->  Bitmap Index Scan on idx_clustering_year  (cost=0.00..525.14 rows=48095 width=0)
                                                (actual time=1.691..1.691 rows=50803 loops=1)
        Index Cond: ((year)::text = '2000'::text)
Planning Time: 0.166 ms
Execution Time: 52.764 ms
```

First, a Bitmap Index Scan is performed on the year attribute of the publ table, followed by a Bitmap Heap Scan.

Non-Clustering Hash Index

Note: Make sure the data is not physically ordered by the indexed attributes due to the clustering index that you created before.

Point Query Repeat the following query multiple times with different conditions for pubID.

```
SELECT * FROM Publ WHERE pubID = ...
```

Which conditions did you use?

For each query a random existing pubID was chosen.

Show the runtime results and compute the throughput.

Executed Queries: 2369970
Queries per Second: 39499.5

Query plan (for one of the queries):

```
Index Scan using idx_clustering_pubid on publ  (cost=0.00..8.02 rows=1 width=112)
                                                (actual time=0.015..0.016 rows=1 loops=1)
  Index Cond: ((pubid)::text = 'books/acm/kim95/Kim95'::text)
Planning Time: 0.122 ms
Execution Time: 0.025 ms
```

For this plan an index scan is again performed on the pubID attribute of the publ table.

Multipoint Query vs. Multipoint Query IN-Predicate – Low Selectivity Repeat the following query multiple times with different conditions for booktitle.

```
SELECT * FROM Publ WHERE booktitle = ...
```

```
SELECT * FROM Publ WHERE pubID IN (...)
```

Which conditions did you use?

To maintain low selectivity, we are using random non-empty book titles out of the dataset. Similarly, for the authors, we are choosing three random authors, but do not need to specify the "non-empty" condition, since an author is always given.

Show the runtime results and compute the throughput.

Multipoint Query
Executed Queries: 136126
Queries per Second: 2268.77

Multipoint Query IN-Predicate
Executed Queries: 172501
Queries per Second: 2875.02

Query plan (for one of the queries):

Multipoint Query

```
Bitmap Heap Scan on publ  (cost=5.38..672.03 rows=178 width=112)
                          (actual time=0.013..0.019 rows=34 loops=1)
  Recheck Cond: ((booktitle)::text = 'Modern Database Systems'::text)
  Heap Blocks: exact=1
  ->  Bitmap Index Scan on idx_clustering_booktitle
                            (cost=0.00..5.33 rows=178 width=0)
                            (actual time=0.009..0.009 rows=34 loops=1)
        Index Cond: ((booktitle)::text = 'Modern Database Systems'::text)
Planning Time: 0.120 ms
Execution Time: 0.032 ms
```

Unlike the B$^+$ tree index, the hash index performs a Bitmap Index Scan on `booktitle` and a Bitmap Heap Scan.

Multipoint Query IN-Predicate

```
Nested Loop  (cost=290.53..869.23 rows=72 width=112)
            (actual time=0.216..3.143 rows=294 loops=1)
  ->  HashAggregate  (cost=290.53..291.25 rows=72 width=23)
                    (actual time=0.196..0.248 rows=294 loops=1)
        Group Key: (auth.pubid)::text
        Batches: 1  Memory Usage: 77kB
        ->  Bitmap Heap Scan on auth
                    (cost=12.56..290.35 rows=72 width=23)
                    (actual time=0.041..0.133 rows=294 loops=1)
"              Recheck Cond: ((name)::text = ANY
                    ('{""William Kent"",""Alfons Kemper"",""Walid G. Aref""}'::text[]))"
              Heap Blocks: exact=5
              ->  Bitmap Index Scan on idx_clustering_name
                        (cost=0.00..12.54 rows=72 width=0)
                        (actual time=0.019..0.020 rows=294 loops=1)
"                    Index Cond: ((name)::text =
                            ANY ('{""William Kent"",""Alfons Kemper"",""Walid G. Aref""}'::text[]
  ->  Index Scan using idx_clustering_pubid on publ
                (cost=0.00..8.02 rows=1 width=112)
                (actual time=0.009..0.009 rows=1 loops=294)
        Index Cond: ((pubid)::text = (auth.pubid)::text)
Planning Time: 0.243 ms
Execution Time: 3.185 ms
```

A Bitmap Heap Scan is performed on the `name` attribute of the `auth` table, followed by hash aggregation. Then, an index scan is performed on `pubID` and finally the results are joined using a nested loop.

Multipoint Query – High Selectivity Repeat the following query multiple times with different conditions for `year`.

```
SELECT * FROM Publ WHERE year = ...
```

Which conditions did you use?

Same as before, we are using random years from the dataset.

Show the runtime results and compute the throughput.

Executed Queries: 2214
Queries per Second: 36.9

Query plan (for one of the queries):

```
Bitmap Heap Scan on publ  (cost=1598.05..24674.51 rows=51877 width=113)
                         (actual time=3.192..72.363 rows=50803 loops=1)
  Recheck Cond: ((year)::text = '2000'::text)
  Heap Blocks: exact=12941
  ->  Bitmap Index Scan on idx_clustering_year
            (cost=0.00..1585.08 rows=51877 width=0)
            (actual time=1.920..1.920 rows=50803 loops=1)
        Index Cond: ((year)::text = '2000'::text)
Planning Time: 0.117 ms
Execution Time: 73.744 ms
```

Here another Bitmap Index Scan with a Bitmap Heap Scan is performed.

Table Scan

Note: Make sure the data is not physically ordered by the indexed attributes due to the clustering index that you created before.

Point Query Repeat the following query multiple times with different conditions for `pubID`.

```
SELECT * FROM Publ WHERE pubID = ...
```

Which conditions did you use?

For each query a random existing `pubID` was chosen.

Show the runtime results and compute the throughput.

Executed Queries: 2080
Queries per Second: 34.67

Query plan (for one of the queries):

```
Gather  (cost=1000.00..29851.09 rows=1 width=113)
        (actual time=0.226..66.255 rows=1 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Seq Scan on publ
        (cost=0.00..28850.99 rows=1 width=113)
        (actual time=24.832..45.490 rows=0 loops=3)
        Filter: ((pubid)::text = 'books/acm/kim95/Kim95'::text)
        Rows Removed by Filter: 411071
Planning Time: 0.627 ms
Execution Time: 66.269 ms
```

First, a parallel Seq Scan is performed on the `publ` table. Then `Gather` is used to collect the results from the two parallel workers, from the parallel Seq Scan. [1]

Multipoint Query vs. Multipoint Query IN-Predicate – Low Selectivity Repeat the following query multiple times with different conditions for `booktitle`.

```
SELECT * FROM Publ WHERE booktitle = ...

SELECT * FROM Publ WHERE pubID IN (...)
```

Which conditions did you use?

To maintain low selectivity, we are using random non-empty book titles out of the dataset. Similarly, for the authors, we are choosing three random authors, but do not need to specify the "non-empty" condition, since an author is always given.

Show the runtime results and compute the throughput.

Multipoint Query
Executed Queries: 1771
Queries per Second: 29.52

Multipoint Query IN-Predicate
Executed Queries: 342
Queries per Second: 5.7

Query plan (for one of the queries):

```
Gather  (cost=1000.00..29868.99 rows=180 width=112)
        (actual time=0.188..75.361 rows=34 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Seq Scan on publ  (cost=0.00..28850.99 rows=75 width=112)
                            (actual time=30.340..54.028 rows=11 loops=3)
        Filter: ((booktitle)::text = ’Modern Database Systems’::text)
        Rows Removed by Filter: 411060
Planning Time: 0.158 ms
Execution Time: 75.373 ms
```

Similar to the previous query, a parallel Seq Scan is performed on the `publ` table, thereafter the results are gathered from the parallel workers.

```
Gather  (cost=45542.30..74465.15 rows=73 width=112)
        (actual time=155.433..236.956 rows=294 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Hash Semi Join  (cost=44542.30..73457.85 rows=30 width=112)
                               (actual time=141.319..216.325 rows=98 loops=3)
        Hash Cond: ((publ.pubid)::text = (auth.pubid)::text)
        ->  Parallel Seq Scan on publ
                (cost=0.00..27566.39 rows=513839 width=112)
                (actual time=0.383..40.797 rows=411071 loops=3)
        ->  Parallel Hash
                (cost=44541.92..44541.92 rows=30 width=23)
                (actual time=139.036..139.036 rows=98 loops=3)
              Buckets: 1024  Batches: 1  Memory Usage: 72kB
              ->  Parallel Seq Scan on auth
                    (cost=0.00..44541.92 rows=30 width=23)
                        (actual time=89.358..138.993 rows=98 loops=3)
"                    Filter: ((name)::text = ANY
                        (’{""William Kent"",""Alfons Kemper"",""Walid G. Aref""}’::text[]))"
                    Rows Removed by Filter: 1031636
Planning Time: 0.690 ms
Execution Time: 236.986 ms
```

For the IN-Predicate query, a Seq Scan is performed on `auth`. A parrallel Hash Semi Join joins the results from the parallel Hash, which we used the Seq Scan for, and a parallel Seq scan on publ. The results are then gathered from the parallel workers, as before.

Multipoint Query – High Selectivity Repeat the following query multiple times with different conditions for `year`.

```
SELECT * FROM Publ WHERE year = ...
```

Which conditions did you use?

Same as before, we are using random years from the dataset.

Show the runtime results and compute the throughput.

Executed Queries: 1069
Queries per Second: 17.82

Query plan (for one of the queries):

```
Gather  (cost=1000.00..35038.69 rows=51877 width=112)
        (actual time=0.221..77.570 rows=50803 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Seq Scan on publ  (cost=0.00..28850.99 rows=21615 width=112)
                                 (actual time=0.024..55.584 rows=16934 loops=3)
        Filter: ((year)::text = '2000'::text)
        Rows Removed by Filter: 394137
Planning Time: 0.117 ms
Execution Time: 78.799 ms
```

A parallel Seq Scan is performed on the `publ` table and then the results from the two workers are gathered.

Discussion

Give the throughput of the query types and index types in queries/second.

	clustering	non-clust. B^+ tree	non-clust. hash	table scan
point (`pubID`)	33089.12	32885.12	39499.50	34.67
multipoint (`booktitle`)	2434.23	2465.50	2268.77	29.52
multipoint-IN (`pubID`)	1371.95	1339.27	2875.02	5.70
multipoint (`year`)	41.05	41.37	36.90	17.82

Discuss the runtime results for the different index types and the table scan. Are the results expected? Why (not)?

Firstly, the table scan is the slowest method for all query types, which is the expected result, as there is no index at all.

Time Spent on this Assignment

Time in hours per person: **XXX**

References

References

[1] May 2024. URL: https://www.postgresql.org/docs/current/how-parallel-query-works.html.