Database Programming Project
Milestone 1 Report

Team Members:

Sarah El-Feel 10002238 T6
Alaa Ashraf 10005152 T2
Abdelrahman Elnagar 10006921 T2

11.11.2024

Query: 1

```
SELECT * FROM data_src ds INNER JOIN datsrcln dl ON ds.datasrc_id =
dl.datasrc_id INNER JOIN nut_data nd ON dl.ndb_no = nd.ndb_no AND dl.nutr_no
= nd.nutr_no;
```

Attribute:

Index on `(nutr_no, ndb_no)` in `datsrcln`

Most expedient index:

```
CREATE INDEX idx_datsrcln_ndb_no_nutr_no ON datsrcln(ndb_no, nutr_no);
```

Effect of the index:

Running 1k	Before	After
Total plan time	932	930
Total exec time	223408	222060
Tps	4.36	4.38
Latency	229	227
Index scans	data_src pkey = 4000 Datasrcln pkey = 3000 nut_data pkey = 3000	data_src pkey = 4000 Datasrcln pkey = 1000 nut_data pkey = 3000 idx_datsrcln_ndb_no_nutr_no = 2000
Query Planner	https://explain.dalibo.com/plan/4c9cb1f079ae0fd4	https://explain.dalibo.com/plan/47gag5g83d3ce225

Justification:

- **Choice of attribute and index:** Since the query only joins tables together with no filtering or aggregates, there was no obvious choice for an index. However, I decided to create an index on two of the attributes of the composite primary key `(ndb_no, nutr_no)` of datsrcln that are needed in the join to avoid using the index with 3 attributes. I created a B+Tree Index that is efficient for exact value searches and could be of major help when joining large tables.
- **Before vs After comparison:** The index was used as seen in the index scans. However no major improvement regarding tps/latency and execution time. Tps increased by **0.02** and latency decreased by **2 ms**, which is so insignificant when comparing the before vs after. Plan and execution are almost the same. The composite index was used twice per each transaction.
- **Justification:** In essence, the query takes so long because `nut_data` has **250k rows**, while the `datasrcln` has **93k rows**. Joining these two tables together is the most

expensive operation, whether the normal primary key index is used or the composite one. In all cases, there is no index that can speed up the join of the two tables.

The index was used, however no significant improvement was found. It was used during the merge join between `datsrcln` and `nut_data` to merge them on sorted columns. Since the created index includes only the relevant attributes, the query planner opted to use it for the merge join, but there is no difference since the tables themselves are very large and matching **93k** to corresponding records from `nut_data` out of **250k** rows takes much time.

- **Result**: no query optimization

TPS/Latency before:

```
[vm@vbox Desktop]$ pgbench -f query1.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query1.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 229.336 ms
tps = 4.360424 (including connections establishing)
tps = 4.360483 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@vbox Desktop]$ pgbench -f query1.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query1.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 227.900 ms
tps = 4.387887 (including connections establishing)
tps = 4.387919 (excluding connections establishing)
```

Time execution before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl.ndb_no = ...	1000	932.3999040000002	223408.75429699986

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl.ndb_no = nd...	1000	930.3929050000006	222060.3869089999

Index scans before:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	3000	93847000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	3000	253668000	253666000
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	1000	93845000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	3000	253668000	253666000
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0
14	16565	16782	public	datsrcln	idx_datsrcln_ndb_...	2000	2000	0

Query: 2

```
SELECT * FROM data_src ds INNER JOIN datsrcln dl ON ds.datasrc_id =
dl.datasrc_id INNER JOIN nut_data nd ON dl.ndb_no = nd.ndb_no AND dl.nutr_no
= nd.nutr_no WHERE year > 2000 AND year < 2001;
```

Attribute:

`year` in `data_src`

Most expedient index:

```
CREATE INDEX idx_data_src_year ON data_src(year);
```

Effect of the index:

Running 1k	Before	After
Total plan time	744	694
Total exec time	32	7.76
Tps	768	1294
Latency	1.301	0.772
Index scans	data_src pkey = 4000 datasrcln pkey = 2000 nut_data pkey = 2000	data src pkey = 4000 Datasrcln pkey = 2000 nut data pkey = 2000 year = 3000
Query Planner	https://explain.dalibo.com/plan/73962fg98hgg6bh1	https://explain.dalibo.com/plan/g254eg9e23agf546

Justification:

- **Choice of attribute and index:** B-tree indexes are highly efficient for range queries, like `year > 2000 AND year < 2001`. They maintain a sorted order, allowing the database to quickly locate and traverse rows within the specified range. Since we are filtering by year, an index on that attribute would speed up locating specific rows with these values. B+Tree would do these range queries faster than hash index.
- **Before vs After comparison:** The query execution shows significant improvements in execution time, latency, and TPS after applying the index, confirming the B-tree's suitability for this type of filtering. Tps increased from **768** to **1294** (almost 50%), latency decreased by **half**, and execution time decreased from **32** to **7.7**. The index on year was also utilized by query planner.
- **Justification:** There is a significant improvement in tps and execution because the query planner utilized the index to search for the required rows in `data_src` (composed of 366 rows) in time complexity `O(log(n))` instead of `O(n)`. Before, the query planner searched

for values between 2000 and 2001 linearly, which took some time, but after, it opted for the index to search for it in a much faster time. However, the query itself is fast because the filtering operation returns 0 records, which results in no records being joined. The index only optimizes the filtering part of the query, not the joins, but since no records match this filter, there is no need to do the join. The execution order is different than in Query 1, as the query planner first filters then joins tables `data_src` and `datsrcln` instead of joining datsrcln with `nut_data` first. This is done since it was found that the join between `data_src` and `datsrcln` would yield no result, hence there is no need to join two heavy tables like `datsrcln` with `nut_data` if in the end they would be joined with 0 records from `data_src`. Our index helped in speeding up the filtering on year. Looking at the logical plan, instead of calculating a cost and returning an estimate on the number of records returned, there will be 0 cost from the join since our `data_src` table contains 0 records after filtering.

- **Result**: significant query optimization

TPS/Latency before:

```
[vm@vbox Desktop]$ pgbench -f query2.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query2.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 1.301 ms
tps = 768.923231 (including connections establishing)
tps = 769.801539 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@vbox Desktop]$ pgbench -f query2.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query2.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 0.772 ms
tps = 1294.878598 (including connections establishing)
tps = 1297.925596 (excluding connections establishing)
```

Time execution before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl.n...	1000	744.2675620000008	32.934993

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl.ndb...	1000	694.4917679999998	7.767976999999996

Index scans before:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	2000	2000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	2000	2000	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	2000	2000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	2000	2000	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0
14	16559	16785	public	data_src	idx_data_src_year	3000	2000	2000

Query: 3

```
SELECT *  FROM data_src ds INNER JOIN datsrcln dl ON ds.datasrc_id =
dl.datasrc_id INNER JOIN nut_data nd ON dl.ndb_no = nd.ndb_no AND dl.nutr_no
= nd.nutr_no WHERE year > 2000;
```

Attribute:

`Year` in `data_src`

Most expedient index:

```
CREATE INDEX idx_data_src_year ON data_src(year);
```

Effect of the index:

Running 1k	Before	After
Total plan time	1238	1227
Total exec time	129631	126300
Tps	7.44	7.63
Latency	134	130
Index scans	data src pkey = 4000 datsrcln pkey = 5000 nut data pkey = 43836000	data src pkey = 4000 datsrcln pkey = 5000 nut data pkey = 43836000 year = 1000
Query Planner	https://explain.dalibo.com/plan/e29580d6044h362d	https://explain.dalibo.com/plan/ehde2860ge63gc0c

Justification:

- **Choice of attribute and index:** B-tree indexes are highly efficient for range queries, like `year > 2000`. They maintain a sorted order, allowing the database to quickly locate the value 2000 and traverse rows within this specified range. Since we are filtering by year, an index on that attribute would speed up locating specific rows with these values. B+Tree would do these range queries faster than hash index since it is better for ranges.
- **Before vs After comparison:** The query execution shows no improvements (almost insignificant improvement) in execution time, latency, and TPS after applying the index, confirming that the index was not beneficial and suitable for this query. Tps increased from **7.44** to **7.63**, latency decreased by **4**, execution time decreased from **129631** to **126300**. The index on year was also utilized by query planner, once per each query run.
- **Justification:** There is no significant improvement in tps and execution because the query planner did not use the index to fetch tuples or remove tuples. When looking at the size of table `data_src`, we can find that `T(data_src)=366`. The number of records with

`year>2000` are **87**. We are talking about a relatively small table and tuple numbers that do not need the index for filtering. The overhead of creating an index and using it for filtering will be greater than just passing sequentially on year and filtering out the values. In that case, going for a sequential scan would be much more efficient and faster since the size of table is small. Regarding why the query itself is not getting optimized whether index is used or not, the issue lies in the join between `datasrcln` and `nut_data`, in which **43,833** `(datasrcln, data_src)` rows are joined with **43,834** `(nut_data)`. This join is expensive and is the main reason why the query is not performing well. The main issue of the query lies in the heavy joins between the table that cannot be optimized further. The query planner might have looked at the index to decide whether it is useful to choose it or not but decided against using it due to the small size of the table.

- **Result**: no query optimization

TPS/Latency before:

```
[vm@vbox Desktop]$ pgbench -f query3.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query3.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 134.246 ms
tps = 7.449004 (including connections establishing)
tps = 7.449105 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@vbox Desktop]$ pgbench -f query3.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query3.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 130.922 ms
tps = 7.638124 (including connections establishing)
tps = 7.638227 (excluding connections establishing)
```

Time execution before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd o...	1000	1238.0993739999992	129631.35647500024

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on d...	1000	1227.9245259999989	126300.80672200011

Index scans before:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	5000	93847000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	43836000	43836000	43834000
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	5000	93847000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	43836000	43836000	43834000
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0
14	16559	16795	public	data_src	idx_data_src_year	1000	1000	1000

Query: 4

```
SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id =
dl.datasrc_id inner join nut_data nd on dl.ndb_no = nd.ndb_no and dl.nutr_no
= nd.nutr_no where min is not null;
```

Attribute:

`Ndb_no` and `nutr_no` in `nut_data` with partial condition on `min`

Most expedient index:

USED: `CREATE INDEX idx_nut_data_pkeys_partial_min ON nut_data(ndb_no, nutr_no) where min is not null;`

NOT USED: `CREATE INDEX idx_nut_data_min_hash ON nut_data using HASH (min);`

Effect of the index:

Running 1k	Before	After
Total plan time	839	935
Total exec time	94338	86937
Tps	10.25	11.06
Latency	97	90
Index scans	data_src pkey = 4000 datsrcln pkey = 2000 nut_data pkey = 2000	data src pkey = 4000 datsrcln pkey = 2000 nut data pkey = 2000 Partial = 1000
Query Planner	https://explain.dalibo.com/plan/h2h1ed2962hf6a24	https://explain.dalibo.com/plan/f5h014118cae2d03

Justification:

- **Choice of attribute and index:** In this query, a hash index on min would've been the most obvious choice for an index attribute and type, however it was not used as shown in index scans picture 6. For that reason, I chose to create an index on the join attributes which are `ndb_no, nutr_no` and apply a partial index on them where the condition is `min is not null`. That way, only rows with `min is not null` are joined and the rest are ignored. The type is still B+Tree index since it is good for join in which exact query matches are needed.
- **Before vs After comparison:** The query execution shows very small improvements in execution time, latency, and TPS after applying the index. Tps increased from **10.25** to **11** (almost 7%), latency decreased by **7 ms**, and execution time decreased by **7401 ms**. The index on year was also utilized by query planner 1 time per each query.

- **Justification:** Even though the query doesn't show a significant improvement, the index was used by the query planner as it presents an advantage over the traditional primary key index. Instead of having a normal B+Tree index, we added a condition to only filter the ones that have a non-null value in `min`. When looking at the rows returned after each step, after filtering, **24k** rows are returned from **250k** in `nut_data`, for which an index is needed. However, even after applying the index that logically and physically speeds up this operation, the query still doesn't show much improvement due to the join of `datsrcln` on `nut_data` in which the filtered **23k** rows are joined with **93k**, resulting in high cost and expensive join. In essence, the highest cost is in the join with an estimation of :

$$T(W) = \frac{T(datsrcln) * T(filtered\ nut\_data)}{\max\left(V(datsrcln, A), V(nut\_data, A)\right)} = \frac{93k * 24k}{\max\left(73k, 24k\right)} = \frac{93k * 24k}{73k} = 30575$$

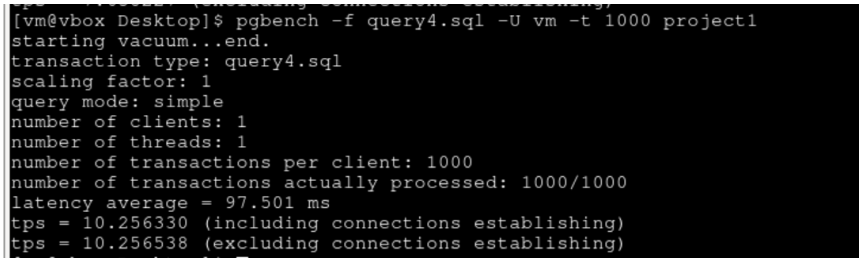
  Where A = common attributes between the two tables. This shows how big the cost is for joining. Here is the cost of filtering:

$$T(W) = \frac{T(nut\_data)}{V(nut\_data, \min)} = \frac{250k}{3294} = 75$$

  Comparing both tells you that cost-based, the query is very slow due to the join itself. It is much more expensive than filtering the rows with values that are not null. Even if we improve the filtering, the cost is still high for the join, for which no optimization can be done since there are already predefined indices on them. The small increase in tps is due to the filtering that was done on `min` attribute, but when comparing it to the whole query, that improvement is insignificant.

- **Extra:** After looking deep in how that query is run, I found out that the query planner creates a bitmap index-on-the-fly using my partial index to be able to filter out the rows in a faster way since we are only creating a bitmap for one single value, making this efficient.
- **Result**: no significant query optimization

TPS/Latency before:



TPS/Latency after:

```
[vm@vbox Desktop]$ pgbench -f query4.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query4.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 90.404 ms
tps = 11.061484 (including connections establishing)
tps = 11.061672 (excluding connections establishing)
```

Time execution before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd ...	1000	839.3787720000008	94338.45210100002

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd o...	1000	935.9635039999989	86937.17098699999

Index scans before:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	2000	2000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	2000	2000	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches_pkey	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_pkey	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts_pkey	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	2000	2000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	2000	2000	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0
14	16592	16801	public	nut_data	idx_nut_data_pkeys_partial_min	1000	23992000	0
15	16592	16802	public	nut_data	idx_nut_data_min_hash	0	0	0

Query: 5

```
SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id =
dl.datasrc_id inner join nut_data nd on dl.ndb_no = nd.ndb_no and dl.nutr_no
= nd.nutr_no where min is null;
```

Attribute:

`ndb_no` and `nutr_no` in `nut_data` with partial condition on `min`

Most expedient index:

```
CREATE INDEX idx_nut_data_pkeys_partial_min ON nut_data(ndb_no, nutr_no)
where min is null;
```

Effect of the index:

Running 1k	Before	After
Total plan time	880	984
Total exec time	181110	156256
Tps	5.39	6.22
Latency	185	90
Index scans	data_src pkey = 4000 datsrcln pkey = 2000 nut_data pkey = 2000	data src pkey = 4000 datsrcln pkey = 3000 nut data pkey = 2000 Partial = 1000
Query Planner	https://explain.dalibo.com/plan/a2167b5c39ba9bbc	https://explain.dalibo.com/plan/c0de65112cbcd44c

Justification:

- **Choice of attribute and index:** In this query, a hash index on min would've been the most obvious choice for an index attribute and type, however it was not used as shown in index scans picture 6. For that reason, I chose to create an index on the join attributes which are `ndb_no, nutr_no` and apply a partial index on them where the condition is `min = null`. That way, only rows with `min = null` are joined and the rest are ignored. The type is still B+Tree index since it is good for join in which exact query matches are needed.
- **Before vs After comparison:** The query execution shows very small improvements in execution time, latency, and TPS after applying the index. Tps increased from **10.25** to **11** (almost 7%), latency decreased by **7 ms**, and execution time decreased by **7401 ms**. The index on year was also utilized by query planner 1 time per each query.
- **Justification:** Even though the query doesn't show a significant improvement, the index was used by the query planner as it presents an advantage over the traditional primary key

index. Instead of having a normal B+Tree index, we added a condition to only filter the ones that have a null value in `min`. When looking at the rows returned after each step, after filtering, **230k** rows are returned from **250k** in `nut_data`, for which an index can be helpful. The join of `datsrcln` on `nut_data` in which the filtered **230k** rows are joined with **93k** results in high cost and expensive join. In essence, the highest cost is in the join with an estimation of:

$$T(W) = \frac{T(datsrcln) * T(filtered\ nut\_data)}{\max\left(V(datsrcln, A), V(nut\_data, A)\right)} = \frac{93k * 230k}{\max\left(73k, 230k\right)} = \frac{93k * 230k}{230k} = 93000$$

Where A = common attributes between the two tables. This shows how big the cost is for joining. Here is the cost of filtering:

$$T(W) = \frac{T(nut\_data)}{V(nut\_data, \min)} = \frac{250k}{3294} = 75$$

Comparing both tells you that cost-based, the query is very slow due to the join itself. It is much more expensive than filtering the rows with values that are null. Even if we improve the filtering, the cost is still high for the join, for which no optimization can be done since there are already predefined indices on them. The slight improvement comes from the index itself since we only join on records that have a null value in `min` which can be explained using the index scan table in which we can see that 230k records are read and fetched per each query run.

- **Extra:** After looking deep into how that query is run, I found out that the query planner uses an index scan instead of sequential way, that is why there is a slight improvement in the tps.
- **Result**: no significant query optimization

TPS/Latency before:

```
[vm@vbox Desktop]$ pgbench -f query5.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query5.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 185.368 ms
tps = 5.394661 (including connections establishing)
tps = 5.394727 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@vbox Desktop]$ pgbench -f query5.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query5.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 160.646 ms
tps = 6.224855 (including connections establishing)
tps = 6.224928 (excluding connections establishing)
```

Time execution before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd o...	1000	880.8159799999995	181110.2705930001

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd o...	1000	984.0084689999998	156256.0243250002

Index scans before:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_bran...	pgbench_branches_pkey	0	0	0
2	16515	16530	public	pgbench_tellers	pgbench_tellers_pkey	0	0	0
3	16518	16532	public	pgbench_acco...	pgbench_accounts_pkey	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	2000	2000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	2000	2000	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_bran...	pgbench_branches_pkey	0	0	0
2	16515	16530	public	pgbench_tellers	pgbench_tellers_pkey	0	0	0
3	16518	16532	public	pgbench_acco...	pgbench_accounts_pkey	0	0	0
4	16559	16616	public	data_src	data_src_pkey	4000	4000	4000
5	16565	16618	public	datsrcln	datsrcln_pkey	3000	93847000	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	2000	2000	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0
14	16592	16809	public	nut_data	idx_nut_data_pkeys_partial_min	1000	229674000	229674000
15	16592	16810	public	nut_data	idx_nut_data_min_hash	0	0	0

Query: 6

```
select * from nutr_def nu inner join nut_data nd on nu.nutr_no = nd.nutr_no
where max is null;
```

Attribute:

`max` in `nut_data`

Most expedient index:

```
NOT USED: CREATE INDEX idx_nut_data_pkeys_partial ON nut_data(nutr_no) where
max is null;
NOT USED: CREATE INDEX idx_nut_data_max_1 ON nut_data(max);
NOT USED: CREATE INDEX idx_nut_data_max_2 ON nut_data USING hash(max);
```

Effect of the index:

Running 1k	Before	After
Total plan time	170	172
Total exec time	217815	220641
Tps	4.38	4.44
Latency	218	224
Index scans	nut_def_pkey=2000	nut_def_pkey=2000
Query Planner	https://explain.dalibo.com/plan/772ggg3c4b62h9d9	https://explain.dalibo.com/plan/623301aceb56563f

Justification:

- **Choice of attribute and index:** In all indexes, the attribute chosen for partial index or normal index was max since we want to get rows with `max = null`. Hash indexes and partial indexes support exact-match queries. For partial, we thought that having a condition would make the join easier as we only join the tables that have `max = null` to reduce number of tuples.
- **Before vs After comparison:** The query execution shows almost no improvements in tps, latency, execution time, and index scans. The differences are minimal since they almost all have the same values. Tps increased by **0.06**, latency by **6 ms**, execution by **2826 ms** and plan by **2 ms**. No other index was used other than `nut_def_pkey`. The differences are minimal when looking at the actual values of these statistics.
- **Justification:** The query does a single join and then a filter on max column. Even if indices usually would've improved a filter operation, in that case the query planner didn't use any of the indexes since the number of null values vs non-null values are **230k** to **23k**. For that

reason, the overhead of creating an index and fetching **230k** rows with the index is greater than sequentially going through the table and checking each value one by one. The query itself takes much time because the join between `nut_data` and `nutr_def` is expensive as we are joining **230k** record with **136**. Even the partial index didn't help because it is a low-selectivity query with many rows returned.

Cost of join:

$$T(W) = \frac{T(filtered\ nut\_data) * T(nutr\_def)}{\max\left(V(nut\_data, A), V(nut\_def, A)\right)} = \frac{230k * 136}{\max\left(136, 136\right)} = \frac{230k * 136}{136} = 230k$$

Cost of filtering:

$$T(W) = \frac{T(nut\_data)}{V(nut\_data, \max)} = \frac{250k}{3703} = 67$$

This tells us that no matter the optimization we have on the filtering, the join is still expensive and cannot be optimized further.

- **Result**: no query optimization

TPS/Latency before:

```
[vm@vbox Desktop]$ pgbench -f query6.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query6.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 227.863 ms
tps = 4.388596 (including connections establishing)
tps = 4.388631 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@vbox Desktop]$ pgbench -f query6.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query6.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 224.877 ms
tps = 4.446878 (including connections establishing)
tps = 4.446912 (excluding connections establishing)
```

Time execution before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * from nutr_def nu inner join nut_data nd on nu.nutr_no = nd.nutr_no where max is null	1000	170.02362399999998	217815.52965499993

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * from nutr_def nu inner join nut_data nd on nu.nutr_no = nd.nutr_no where max is null	1000	172.90156699999986	220641.719231

Index scans before:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	0	0	0
5	16565	16618	public	datsrcln	datsrcln_pkey	0	0	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	0	0	0
10	16598	16628	public	nutr_def	nutr_def_pkey	2000	2000	2000
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	0	0	0
5	16565	16618	public	datsrcln	datsrcln_pkey	0	0	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	0	0	0
10	16598	16628	public	nutr_def	nutr_def_pkey	2000	2000	2000
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0
14	16592	16813	public	nut_data	idx_nut_data_pkey...	0	0	0
15	16592	16814	public	nut_data	idx_nut_data_max_1	0	0	0
16	16592	16815	public	nut_data	idx_nut_data_max_2	0	0	0

Query: 7

```
select * from nutr_def nu inner join nut_data nd on nu.nutr_no = nd.nutr_no
where max is null order by max;
```

Attribute:

`max` in `nut_data`

Most expedient index:

```
NOT USED: CREATE INDEX idx_nut_data_pkeys_partial ON nut_data(nutr_no) where
max is null;
NOT USED: CREATE INDEX idx_nut_data_max_1 ON nut_data(max);
NOT USED: CREATE INDEX idx_nut_data_max_2 ON nut_data USING hash(max);
NOT USED: CREATE INDEX idx_nut_data_max_3 ON nut_data USING hash(max) WHERE
max IS NULL;
```

Effect of the index:

Running 1k	Before	After
Total plan time	196	226
Total exec time	344789	350424
Tps	2.82	2.78
Latency	353	359
Index scans	nut_def_pkey=2000	nut_def_pkey=2000
Query Planner	https://explain.dalibo.com/plan/1f4f5c8cba35fg6e	https://explain.dalibo.com/plan/2c85aa1f16778f3g

Justification:

- **Choice of attribute and index:** In all indexes, the attribute chosen for partial index or normal index was max since we want to get rows with `max = null` and then order them by that column. Hash indexes and partial indexes support exact-match queries and that is why we chose them. For partial, we thought that having a condition would make the join easier as we only join the tables that have `max = null` to reduce number of tuples.

- **Before vs After comparison:** The query execution shows almost no improvements in tps, latency, execution time, and index scans. The differences are minimal since they almost all have the same values. Tps decreased by **0.04**, latency increased by **6 ms**. The only major difference is plan time (increased by **30 ms**) since the query planner might have needed more time to look at the many indices we created and choose the optimal plan.

- **Justification:** The query does a single join and then a filter on max column. Even if indices usually would've improved a filter operation, in that case the query planner didn't use any of

the indexes since the number of null values vs non-null values are **230k** to **23k**. For that reason, the overhead of creating an index and fetching **230k** rows with the index is greater than sequentially going through the table and checking each value one by one. The query itself takes much time because the join between `nut_data` and `nutr_def` is expensive as we are joining **230k** record with **136**. The cost for join and filter are same as in query 6 and this tells us that no matter the optimization we have on the filtering, the join is still expensive and cannot be optimized further. The query also is relatively slower than query 6 because of the unnecessary order by max. When all values in the sorted column are `NULL`, PostgreSQL will still complete the sorting operation. However, since all values are the same (`NULL`), the result order for that column won't visibly change. It will effectively appear as if nothing was sorted by that column, but the sorting step has still technically been executed.

- **Result**: no query optimization

TPS/Latency before:

```
[vm@vbox Desktop]$ pgbench -f query7.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query7.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 353.384 ms
tps = 2.829784 (including connections establishing)
tps = 2.829796 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@vbox Desktop]$ pgbench -f query7.sql -U vm -t 1000 project1
starting vacuum...end.
transaction type: query7.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 359.266 ms
tps = 2.783453 (including connections establishing)
tps = 2.783470 (excluding connections establishing)
```

Time execution before:

```
10  SELECT *
```

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * from nutr_def nu inner join nut_data nd on nu.nutr_no = nd.nutr_no where max is null order by ...	1000	196.09985600000022	344789.0226740002

Time execution after:

```
10  SELECT *
```

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * from nutr_def nu inner join nut_data nd on nu.nutr_no = nd.nutr_no where max is null order by max	1000	226.57108200000033	350424.2630899999

Index scans before:

```
where max is null order by max;
```

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	0	0	0
5	16565	16618	public	datsrcln	datsrcln_pkey	0	0	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	0	0	0
10	16598	16628	public	nutr_def	nutr_def_pkey	2000	2000	2000
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	0	0	0
5	16565	16618	public	datsrcln	datsrcln_pkey	0	0	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	0	0	0
9	16592	16626	public	nut_data	nut_data_pkey	0	0	0
10	16598	16628	public	nutr_def	nutr_def_pkey	2000	2000	2000
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	0	0	0
13	16610	16777	public	weight	idx_weight_comp_2	0	0	0
14	16592	16830	public	nut_data	idx_nut_data_pkey...	0	0	0
15	16592	16831	public	nut_data	idx_nut_data_max_1	0	0	0
16	16592	16832	public	nut_data	idx_nut_data_max_2	0	0	0
17	16592	16833	public	nut_data	idx_nut_data_max_3	0	0	0

Query: 8

```
SELECT food_des.ndb_no,  food_des.long_desc,  gm_wgt
FROM food_des
INNER JOIN weight ON weight.ndb_no=food_des.ndb_no
WHERE weight.msre_desc ='serving';
```

Attribute:

Index on `(msre)` in `weight`

Most expedient index:

```
CREATE INDEX idx_weight_msre_desc ON weight USING hash(msre_desc);
```

Effect of the index:

Running 1k	Before	After
Total plan time	144.6	124
Total exec time	2298.97	1560
Tps	349.84	603.85
Latency	2.858	1.656
Index scans	food_des = 2000 weight_pkey = 2000	food_des = 2000 weight_pkey = 2000 idx_weight_msre_desc = 1000
Query Planner	https://explain.dalibo.com/plan/b0c77a9a8ach8422#plan	https://explain.dalibo.com/plan/b7746h11af38b058

Justification:

- **Choice of attribute and index:** Since the query only joins tables together and then filters by an exact value `weight.msre_desc ='serving'`, I chose to create a Hash index on `msre_desc`. Hash indexes are highly efficient for exact queries. Other index types, such as bitmap are not suitable because there are many distinct values and B+ trees indices are more optimal for range queries but can be a possible option too.
- **Before vs After comparison:** The index was used as seen in the index scans. The query execution shows significant improvements in plan time from **144** to **124** execution time **2298** to **1560** , latency from **2.858** to **1.656**, and Tps from **349** to **603** after applying the index, confirming the Hash Index's suitability for this type of filtering.
- **Justification:** Whether before or after the index, the **7146** rows in `food_des` will be fetched. Before the index, row by row table scan occurs on the weight table and the filter goes through and removes **12590** rows and keeps the required **419** rows. After the index is created, it is used to directly only get the required **419** rows. Then the join happens;

therefore, in this query the index I created improved the performance in the filtering section of the query.

- **Result**: Query optimized
- **Extra:** Physical Plan goes from Sequential Scan on weight to Index Scan

TPS/Latency before:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery8.s
ql -U vm -t 1000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery8.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 2.858 ms
tps = 349.840239 (including connections establishing)
tps = 350.084989 (excluding connections establishing)
[vm@archlinux ~]$
```

TPS/Latency after:

```
transaction type: /home/vm/Desktop/Project1Queries/individualquery8.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 1.656 ms
tps = 603.853173 (including connections establishing)
tps = 604.455662 (excluding connections establishing)
[vm@archlinux ~]$
```

Time execution before:

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * fr...	315	59.151751	97740.80815799994
2	select * fr...	300	52.75743199999998	93285.842656
3	/*pga4da...	2179	4663.1001689999985	25768.339260999997
4	/*pga4da...	801	1718.0514220000002	10072.670525000005
5	SELECT	1000	144.60741199999984	2298.975009000002

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT food_des....	1000	124.479429	1560.4828299999988

Index scans before:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	0	0	0
5	16565	16618	public	datsrcln	datsrcln_pkey	0	0	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	2000	2000	0
9	16592	16626	public	nut_data	nut_data_pkey	0	0	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	2000	2000	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	0	0	0
5	16565	16618	public	datsrcln	datsrcln_pkey	0	0	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	2000	2000	0
9	16592	16626	public	nut_data	nut_data_pkey	0	0	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	2000	2000	0
13	16610	24578	public	weight	idx_weight_msre_...	1000	419000	0

Query: 9

```
SELECT gm_wgt from weight;
```

Attribute:

None but attempted: Index on (gm_wgt) in weight

Most expedient index:

NOT USED: `CREATE INDEX idx_weight_gm_wgt ON weight(gm_wgt);`

Effect of the index:

Running 100k	Before	After
Total plan time	1392	1449
Total exec time	229664	236479
Tps	440.910269	444.144
Latency	2.268	2.252
Index scans	None	None
Query Planner	https://explain.dalibo.com/plan/796c6cd08degg91g#plan	https://explain.dalibo.com/plan/ff380h5c75ga5gf5#plan

Justification:

- **Choice of attribute and index:** Since there is no filtering, no index on any attribute is needed as we are only fetching the whole column in the relation.
- **Before vs After comparison:** The query execution shows no improvements in execution time, latency, and TPS after applying the index and the index created isn't used by the planner/query anyway. The plan time increased from **1392** to **1449**, the Execution time increased from **229664** to **236479**, the Tps increased from **440** to **444**, the Latency decreased from **2.268** to **2.252**. However, these apparent changes are just natural execution variations since clearly the indices are not used and the change in numbers is minimal.
- **Justification:** Since there is no filtering, no index was used even after creating the B+ tree index on the selected column. It also doesn't make sense to use an index in this specific query since the whole column is being chosen, so why check the pointer to every record then to the record instead of directly to all the records in the table in one go. It is more efficient to not use any index in this query.
- **Result**: No Query optimization
- **Extra:** I even tried to make a huge sample table with millions of rows to confirm that the inherent nature of the query itself implies that it will not be optimized by an index, which was confirmed by this experiment when the planner never used any index.

TPS/Latency before:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery9.sql -U vm -t 100000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery9.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 100000
number of transactions actually processed: 100000/100000
latency average = 2.268 ms
tps = 440.910269 (including connections establishing)
tps = 440.914816 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery9.sql -U vm -t 100000 ProjOne
starting vacuum...end.

transaction type: /home/vm/Desktop/Project1Queries/individualquery9.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 100000
number of transactions actually processed: 100000/100000
latency average = 2.252 ms
tps = 444.144799 (including connections establishing)
tps = 444.148606 (excluding connections establishing)
```

Time execution before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT gm_wgt fro...	100000	1392.2812189999815	229664.8448739991

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT gm_wgt fro...	100000	1449.4388149999684	236479.76543300215

Index scans before & after:

None

Query: 10

```
SELECT min(gm_wgt)
FROM weight;
```

Attribute:

Index on `(gm_wgt )` in `weight`

Most expedient index:

```
CREATE INDEX idx_weight_gm_wgt ON weight(gm_wgt);
```

Effect of the index:

Running 100k	Before	After
Total plan time	2611	1779
Total exec time	63795	1046
Tps	1148	6770
Latency	0.871	0.148
Index scans	None	idx_gm_wgt = 100000
Query Planner	https://explain.dalibo.com/plan/b0c77a9a8ach8422#plan	https://explain.dalibo.com/plan/b7746h11af38b058

Justification:

- **Choice of attribute and index:** Since we are aggregating only by `gm_wgt`, an index on that column would make it faster. B-tree indexes are highly efficient for aggregates like `min(gm_wgt)`. They maintain a sorted order, allowing the database to quickly locate and traverse rows within the specified range. Other index types, such as hash, do not support aggregate operations.
- **Before vs After comparison:** The index was used as seen in the index scans. Major improvements occurred regarding tps/latency and execution time. The query execution shows significant improvements in plan time from **2611** to **1779** execution time **63795** to **1046** , latency from **0.871** to **0.148**, and TPS from **1148** to **6770** after applying the index, confirming the B+ Index's suitability for this type of filtering.
- **Justification:** Since we are filtering by `gm_wgt`, an index on that column would make it faster to only select records that are within this range without having to scan the whole table linearly to fetch **13009** rows like before the index so instead we find only the **1** required row in `O(log(n))`.
- **Result:** significant query optimization

- **Extra:** Query plan shows shift from Sequential Scan to Index Only Scan.

TPS/Latency before:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery10.sql -U vm -t 100000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery10.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 100000
number of transactions actually processed: 100000/100000
latency average = 0.871 ms
tps = 1148.407756 (including connections establishing)
tps = 1148.463804 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery10.sql -U vm -t 100000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery10.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 100000
number of transactions actually processed: 100000/100000
latency average = 0.148 ms
tps = 6770.318207 (including connections establishing)
tps = 6771.091241 (excluding connections establishing)
```

Time execution before:

3	select mi...	100000	2611.792614999976	63795.171692000156

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT min(gm_wgt)	100000	1779.9543360001064	1046.76646499998

Index scans before:

None

Index scans after:

12	16929	16951	public	weight	weight_pkey	0	0	0
13	16929	17032	public	weight	idx_weight_gm_wgt	100000	100000	0

Query: 11

```
SELECT min(gm_wgt), msre_desc FROM food_des
INNER JOIN weight ON weight.ndb_no=food_des.ndb_no GROUP BY msre_desc;
```

Attribute:

`None but attempted:` Index on `(gm_wgt )` in `weight & (msre_desc)` in `weight`

Most expedient index:

NOT USED:

```
1  create index idx_weight_gm_wgt on weight(gm_wgt);
2  create index idx_weight_msre_desc on weight(msre_desc);
3  create index idx_weight_comp on weight(msre_desc, gm_wgt)
4  create index idx_weight_comp_2 on weight(gm_wgt, msre_desc)
```

Effect of the index:

Running 100k	Before	After
Total plan time	13563	13103
Total exec time	429231	454438
Tps	214	220
Latency	4.67	4.54
Index scans	food_des_pkey = 300000 weight_pkey =200000	food_des_pkey = 300000 weight_pkey =200000
Query Planner	https://explain.dalibo.com/plan/dfg20f11417b14gb	https://explain.dalibo.com/plan/4ehc3dcec39a2c85

Justification:

- **Choice of attribute and index:** Since the query only joins tables together with aggregate MIN on `gm_wgt` and since there is an aggregate GROUP BY on `msre` I created several indices on them (`gm_wgt` & `msre`) I created a B+Tree Index that is efficient for aggregates which is the case here. There are already default indices on ndb_no in both tables so I didn't create any new ones on them.
- **Before vs After comparison:**The query execution shows no improvements in execution time, latency, and TPS after applying the index and the indices created aren't used by the planner/query anyway. The plan time decreased from **13563** to **13103**, the Execution time increased from **429231** to **454438**, the Tps increased from **214** to **220** and latency decreased from **4.67** to **4.54**. However, these apparent improvements are just natural

execution variations since clearly the indices are not used and the change in numbers is minimal.

- **Justification:** The major cost in this query is the join of the 13k rows in weight and 7k rows in `food_des`. There are already the needed indices on their primary keys to optimize this part of the query and there are no other better indices that can be created to improve the performance of the join. The reason why the indices of the `gm_wgt` and `msre_desc` are not used is most likely because the way the database handles these operations in this specific case is better than using an index. Check 'Extra' for more details.
- **Result**: No Query optimization
- **Extra:** Query Plan does not change after creating any index and includes no index scans except on the default primary key index of `food_des`. We wanted to understand further why the indices of the `gm_wgt` and `msre_desc` are not used, we then found that the plan uses a Hash aggregate which automatically most likely handles both of `GROUP BY msre_desc and min(gm_wgt)`. Whenever a new record is added to the bucket, the minimum is adjusted automatically using the Hash. So there is no need to create an index on `gm_wgt`.

TPS/Latency before:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery11.sql -U vm -t 100000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery11.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 100000
number of transactions actually processed: 100000/100000
latency average = 4.672 ms
tps = 214.062996 (including connections establishing)
tps = 214.063890 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery11.sql -U vm -t 100000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery11.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 100000
number of transactions actually processed: 100000/100000
latency average = 4.542 ms
tps = 220.158144 (including connections establishing)
tps = 220.160021 (excluding connections establishing)
[vm@archlinux ~]$
```

Time execution before:

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select mi...	100000	13563.734250999954	429231.4411109983

Time execution after:

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	SELECT min(gm_wgt...	100000	13103.694316000041	454438.47668400494

Index scans before:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	0	0	0
5	16565	16618	public	datsrcln	datsrcln_pkey	0	0	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	300000	714800000	0
9	16592	16626	public	nut_data	nut_data_pkey	0	0	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	200000	200000	0
13	16610	24600	public	weight	idx_weight_gm_wgt	0	0	0

Index scans after:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	16521	16528	public	pgbench_br...	pgbench_branches...	0	0	0
2	16515	16530	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
3	16518	16532	public	pgbench_ac...	pgbench_accounts...	0	0	0
4	16559	16616	public	data_src	data_src_pkey	0	0	0
5	16565	16618	public	datsrcln	datsrcln_pkey	0	0	0
6	16568	16620	public	deriv_cd	deriv_cd_pkey	0	0	0
7	16574	16622	public	fd_group	fd_group_pkey	0	0	0
8	16580	16624	public	food_des	food_des_pkey	300000	714800000	0
9	16592	16626	public	nut_data	nut_data_pkey	0	0	0
10	16598	16628	public	nutr_def	nutr_def_pkey	0	0	0
11	16604	16630	public	src_cd	src_cd_pkey	0	0	0
12	16610	16632	public	weight	weight_pkey	200000	200000	0
13	16610	24600	public	weight	idx_weight_gm_wgt	0	0	0

Query: 12

```
select count(*) from
(select min(gm_wgt), msre_desc from food_des
INNER JOIN weight ON weight.ndb_no=food_des.ndb_no group by msre_desc) as t;
```

Attribute:

`None but attempted:` Index on `(gm_wgt )` in `weight & (msre)` in `weight`

Most expedient index:

NOT USED:

```
1  create index idx_weight_gm_wgt on weight(gm_wgt);
2  create index idx_weight_msre_desc on weight(msre_desc);
3  create index idx_weight_comp on weight(msre_desc, gm_wgt)
4  create index idx_weight_comp_2 on weight(gm_wgt, msre_desc)
```

Effect of the index:

Running 10k	Before	After
Total plan time	1604, 1264	1333
Total exec time	39157, 35971	38873
Tps	222, 253	236
Latency	4.489, 3.9	4.229
Index scans	food_des_pkey = 30000 weight_pkey =20000	food_des_pkey = 30000 weight_pkey =20000
Query Planner	https://explain.dalibo.com/plan/gf3cfdchcbg23ab7	https://explain.dalibo.com/plan/9836982d64f39e85

Justification:

- **Choice of attribute and index:** Since the query only joins tables together with aggregate MIN on `gm_wgt` and since there is an aggregate GROUP BY on `msre` I created several indices on them (`gm_wgt` & `msre`) I created a B+Tree Index that is efficient for aggregates, which is the case here. There are already default indices on ndb_no in both tables so I didn't create any new ones on them.
- **Before vs After comparison:**The plan time decreased from **1604** to **1333**, the Execution time decreased from **39157** to **38873**, the Tps value increased from **222** to **236**, the Latency value decreased from **4.489** to **4.229**. I noticed that the numbers seem to indicate an improvement; therefore, I decided to make another run without the index and noticed that the Plan time, Exec Time, Tps, and Latency were **1264**, **35971**, **253**, **3.9** accordingly, all of

which were better numbers than the first run without the index and most of the time better or just like the statistics after the index. In conclusion, the query execution shows no improvements in execution time, latency, and TPS after applying the index and the index created isn't used by the planner. These are normal fluctuations since clearly even before adding the index, running multiple times slightly 'improves' the statistics then after the index slightly 'worsens' but it's just a normal fluctuation.

- **Justification:** The major cost in this query is the join of the 13k rows in weight and 7k rows in `food_des`. There are already the needed indices on their primary keys to optimize this part of the query and there are no other better indices that can be created to improve the performance of the join. The reason why the indices of the `gm_wgt` and `msre_desc` are not used is most likely because the way the database handles these operations in this specific case is better than using an index. This is the same as query 11 but here all what happens is after finishing all this we get the count of the resulting table. Because the internal table is not optimized by these indices anyway, so will getting its count. Check 'Extra' for more details.
- **Result**: No Query optimization
- **Extra:** Query Plan does not change after creating any index and includes no index scans except on the default primary key index of `food_des`. We wanted to understand further why the indices of the `gm_wgt` and `msre_desc` are not used, we then found that the plan uses a Hash aggregate which automatically most likely handles both of `GROUP BY msre_desc and min(gm_wgt)`. Whenever a new record is added to the bucket, the minimum is adjusted automatically using the Hash. So there is no need to create an index on `gm_wgt`. As mentioned this is the same as Query 11 and the fact that so far this is not optimized, getting the count of the resulting relation will surely not get optimized either.

TPS/Latency before:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery12.sql -U vm -t 10000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery12.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 4.489 ms
tps = 222.752325 (including connections establishing)
tps = 222.761436 (excluding connections establishing)
```

TPS/Latency after:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery12.sql -U vm -t 10000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery12.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 4.229 ms
tps = 236.463015 (including connections establishing)
tps = 236.472696 (excluding connections establishing)
```

Time execution before:

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select co...	10000	1604.8558830000054	39157.68391500007

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select co...	10000	1264.799404000003	35971.03954500012

Time execution after:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select count(*) from	10000	1333.7598190000012	38873.20477700013

Index scans before:

8	16899	16943	public	food_des	food_des_pkey	30000	71480000	0
9	16911	16945	public	nut_data	nut_data_pkey	0	0	0
10	16917	16947	public	nutr_def	nutr_def_pkey	0	0	0
11	16923	16949	public	src_cd	src_cd_pkey	0	0	0
12	16929	16951	public	weight	weight_pkey	20000	20000	0

Index scans after:

8	16899	16943	public	food_des	food_des_pkey	30000	71480000	0
9	16911	16945	public	nut_data	nut_data_pkey	0	0	0
10	16917	16947	public	nutr_def	nutr_def_pkey	0	0	0
11	16923	16949	public	src_cd	src_cd_pkey	0	0	0
12	16929	16951	public	weight	weight_pkey	20000	20000	0

Query: 13

```
select min(gm_wgt), msre_desc
from weight
where gm_wgt > 100
group by msre_desc;
```

Attribute:

Composite: `gm_wgt` THEN `msre_desc` (order mattered)

Most expedient index:

```
CREATE INDEX idx_weight_comp_2 on weight(gm_wgt, msre_desc);
```

```
create index idx_weight_gm_wgt on weight(gm_wgt);
create index idx_weight_msre_desc on weight(msre_desc);
create index idx_weight_comp on weight(msre_desc, gm_wgt)
create index idx_weight_comp_2 on weight(gm_wgt, msre_desc)

```

Effect of the index:

Running 10k	Before	After
Total plan time	446	283
Total exec time	16179	9814
Tps	521	837
Latency	1.919	1.194
Index scans	None	None, None, idx_weight_comp_2 = 10000
Query Planner	https://explain.dalibo.com/plan/2ag007fg7c2d892g	https://explain.dalibo.com/plan/aa07916aga99a57c

Justification:

Composite index on gm_wgt THEN msre_desc (order mattered) is the optimal choice because:

- **Choice of attribute and index:** There is a need to filter by and get an aggregate of `gm_wgt` and a need to order by `msre_desc`. B+ Tree is most suitable for range and aggregate queries. Composite Index is most useful because of operations on multiple attributes.
- **Before vs After comparison:** Most indices used made no difference and were never used, however index `idx_weight_comp_2` showed a large optimization in Plan time from **446** to **283** , in Execution time from **16179** to **9814**, in Tps from **521** to **837**, in Latency from **1.919**

to **1.194**. Which proves the efficiency of the composite B+ index on `weight(gm_wgt, msre_desc).`

- **Justification:** The reason why the order of the attribute in the index creation matters is that in this query, `where gm_wgt > 100` is done before `group by msre_desc`, which means that when looking for a suitable index, we want the attribute `gm_wgt` to be the initial attribute in the composite index after which we will use the secondary attribute in the index which is `msre_desc` to Group By. Since we are filtering by `gm_wgt`, an index on that column would make it faster to only select the estimated **5853** records that are within this range without having to scan the whole table linearly row by row for **7156** rows. This improves the search from `O(n)` to `O(log(n))`, and it will greatly reduce the number of database record fetches as explained above. Then, when the group by is done, the index also helps find the groups using the index directly instead of one by one.
- **Result**: Query optimized.
- **Extra:** We can see in the Query plan that the Seq Scan and filter of `where gm_wgt > 100` is done before the Hash Aggregate `group by msre_desc,` which explains the planner's behavior in choosing only the index created and used above.

Latency & TPS Before:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery13.sql -U vm -t 10000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery13.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 1.919 ms
tps = 521.226180 (including connections establishing)
tps = 521.349372 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery13.sql -U vm -t 10000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery13.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 1.194 ms
tps = 837.417422 (including connections establishing)
tps = 837.592955 (excluding connections establishing)
[vm@archlinux ~]$
```

Plan and Execution Time Before:

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select mi...	10000	446.93606400000044	16179.722915000006

Plan and Execution Time After:

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select mi...	10000	283.1070140000006	9814.943091999994

Index Scan Before:

12	16929	16951	public	weight	weight_pkey	0	0	0
13	17053	17057	public	test	test_pkey	0	0	0

Index Scan After:

13	17053	17057	public	test	test_pkey	0	0	0
14	16929	17084	public	weight	idx_weight_comp_2	10000	58530000	0

Query: 14

```
select gm_wgt
from weight
where gm_wgt > 100;
```

Attribute:

Index on `gm_wgt` in `weight`

Most expedient index:

```
CREATE INDEX idx_weight_gm_wgt ON weight(gm_wgt);
```

Effect of the index:

Running 10k	Before	After
Total plan time	353	224
Total exec time	17551	8213
Tps	493	966
Latency	2.025	1.035
Index scans	None	idx_weight_gm_wgt = 10000
Query Planner	https://explain.dalibo.com/plan/hgeeba4f47fb10dg	https://explain.dalibo.com/plan/33ccb87f9e54e22f

Justification:

- **Choice of attribute and index:** There is a range query `WHERE gm_wgt > 100` ; therefore, I chose to create a B+ Index on `gm_wgt`. B-tree indexes are highly efficient for range queries, like `WHERE gm_wgt > 100`. They maintain a sorted order, allowing the database to quickly locate and traverse rows within the specified range. Other index types, such as hash, do not support range operations.
- **Before vs After comparison:** Index `idx_weight_gm_wgt` showed a decrease in Plan time from **353** to **224**. However, there are large optimizations in Execution time from **17551** to **8213**, in Tps from **493** to **996**, in Latency from **2.025** to **1.035**. Which proves the efficiency of the composite B+ index on `weight(gm_wgt).`
- **Justification:** Since we are filtering by `gm_wgt`, an index on that column would make it faster to only select the **5853** records that are within this range without having to scan the whole table linearly row by row for **7156** rows. This improves the search from `O(n)` to `O(log(n))`, and it will greatly reduce the number of database record fetches as explained above.

- **Result**: Query optimized.
- **Extra:** It is clear from the Query Plan that it was a Sequential Scan and then it became an Index scan with lower cost.

Latency & TPS Before:

```
[vm@archlinux ~]$ pgbench -f /home/vm/Desktop/Project1Queries/individualquery14.sql -U vm -t 10000 ProjOne
starting vacuum...end.
transaction type: /home/vm/Desktop/Project1Queries/individualquery14.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 2.025 ms
tps = 493.781102 (including connections establishing)
tps = 493.846673 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@vbox Desktop]$ pgbench -f query14.sql -U vm -t 10000 project1
starting vacuum...end.
transaction type: query14.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 1.035 ms
tps = 966.447468 (including connections establishing)
tps = 966.649027 (excluding connections establishing)
```

Plan and Execution Time Before:

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select g...	10000	353.3077150000009	17551.708763000028

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select gm_wgt	10000	224.34814599999956	8213.235976000002

Index Scan Before:

None

Index Scan After:

14	17053	17059	public	test	gm_indx	0	0	0
15	16929	17065	public	weight	idx_weight_gm_wgt	10000	58530000	0

Query: 15

```
Select * from src_cd sc
Inner join nut_data nd on sc.src_cd = nd.src_cd
Inner join food_des fd on fd.ndb_no = nd. ndb_no
Where sc.src_cd = 2;
```

Attribute:

Index on `ndb_no` in `nut_data` table
Index on `src_cd` in `nut_data` table

Most expedient index:

USED: `CREATE INDEX idx_nut_data_ndb_no ON nut_data (ndb_no);`
NOT USED: `CREATE INDEX idx_nut_data_src_cd_hash ON nut_data using hash (src_cd);`

Effect of the index:

Running 10k	Before	After
Total plan time	2,855.92	2,977.97
Total exec time	123.85	133.08
Tps	1679.67	1570.93
Latency	0.596	0.637
Index scans		idx_nut_data_ndb_no = 20000 idx_nut_data_src_cd_hash = 0
Query Planner	https://explain.dalibo.com/plan/f32f378fa5d1dcd1#plan	https://explain.dalibo.com/plan/6b7egcgbghf9h35c

Justification:

- **Attribute and Index Choice:** using a hash index on `src_cd` was appropriate for direct equality checks `(WHERE src_cd = 2)` to be efficient for exact-match queries and to avoid the overhead of scanning large datasets and it was selected by the query planner but not scanned as the table is not large. B-tree index on `ndb_no` allowed faster joins between `nut_data` and `food_des`, leveraging the equality condition (`ON fd.ndb_no = nd.ndb_no)` to try to check if the single attribute will work out better and the choice of B-tree helps in Range queries and joins.

- **Before vs after Comparison:** There is no improvement regarding the performance of the query where the tps went from **1679** to **1570.93** and latency from **0.596** to **0.637** with a very insignificant change resulting in no improvement.
- **Justification:**

  Already indexed Join: In the join between the three tables all of their primary keys already have their index and when we try to make index on `ndb_no` in table `nut_data` to see if single attribute index is better than the composite there was no effect on performance despite the usage of the index.

  Zero Rows condition: Having the condition `src_cd = 2` resulting in zero rows from the table results in the rest of operations never executing because doing the join with the other side having a zero output makes it impossible to show any effect of adding indexes.

  Small Table: Indexes on `src_cd` using hash instead of btree was not even used and didn't make any optimizations as the table `src_cd` only contains 10 rows and any index on it will cause overhead.
- **Result:** No significant query optimization

Latency & TPS Before:

```
[vm@archlinux Desktop]$  pgbench -f Query15 -U vm -t 10000 project
starting vacuum...end.
transaction type: Query15
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 0.596 ms
tps = 1677.605617 (including connections establishing)
tps = 1679.675760 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux Desktop]$  pgbench -f Query15 -U vm -t 10000 project
starting vacuum...end.
transaction type: Query15
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 0.637 ms
tps = 1569.578476 (including connections establishing)
tps = 1570.932050 (excluding connections establishing)
```

Plan and Execution Time Before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * fr...	10000	2855.927976000014	123.85133599999972

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * from src_cd ...	10000	2977.979670000005	133.08447800000025

Index Scan After:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
5	16544	16588	public	food_des	food_des_pkey	40010	40010	0
6	16556	16590	public	nut_data	nut_data_pkey	20010	20010	0
7	16562	16592	public	nutr_def	nutr_def_pkey	0	0	0
8	16568	16594	public	src_cd	src_cd_pkey	0	0	0
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_bran...	pgbench_branches_pkey	0	0	0
11	16682	16697	public	pgbench_tellers	pgbench_tellers_pkey	0	0	0
12	16685	16699	public	pgbench_acco...	pgbench_accounts_pkey	0	0	0
13	16556	32770	public	nut_data	idx_nut_data_src_cd	0	0	0
14	16568	32772	public	src_cd	idx_src_cd_src_cd_hash	0	0	0
15	16556	32773	public	nut_data	idx_nut_data_src_cd_hash	0	0	0
16	16556	32774	public	nut_data	idx_nut_data_ndb_no	20000	20000	0

Query: 16

```
Select * from src_cd sc
Inner join nut_data nd on sc.src_cd = nd.src_cd
Inner join food_des fd on fd.ndb_no = nd. ndb_no
```

Attribute:

Btree on `ndb_no` in `nut_data` table

Most expedient index:

```
CREATE INDEX idx_nut_data_ndb_no ON nut_data (ndb_no);
```

Effect of the index:

Running 10k	Before	After
Total plan time	2,434.56	2,797.65
Total exec time	3,317,774.80	3,564,036.04
Tps	2.89	2.70
Latency	345.61	369.679
Index scans		idx_nut_data_ndb_no = 20,000
Query Planner	https://explain.dalibo.com/plan/ea6bg50e04e99e4g#plan	https://explain.dalibo.com/plan/71ghg372bdg11a0c

Justification:

- **Attribute and Index Choice:** B-tree index on `ndb_no` allowed faster joins between `nut_data` and `food_des`, leveraging the equality condition (`ON fd.ndb_no = nd.ndb_no`) trying to check if the single attribute will work out better despite the fact that hash was tried but not scanned.
- **Before vs After Comparison:** There is no improvement regarding the performance of the query where the tps went from **2.89** to **2.70** and latency from **345.61** to **369.679** with a very insignificant change resulting in no improvement.
- **Justification:**

  Already indexed Join: In the join between the three tables all of them already have their own indexes and when we try to make an index on `ndb_no` in table `nut_data` to see if a single attribute index is better than the composite there was no effect on performance despite the usage of the index.

  No conditions: Having no conditions at all sent a sign of no selectivity to the tables being requested leaving no place for optimization for the joins which already indexed.

- **Result:** no significant query optimization
- **Extra:** This query is different in execution than the one before where the query planner uses the hash inner join resulting in creating hash tables for the data on the joining attributes in order to help while scanning.

Latency & TPS Before:

```
[vm@archlinux Desktop]$  pgbench -f Query16 -U vm -t 10000 project
starting vacuum...end.
transaction type: Query16
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 345.610 ms
tps = 2.893434 (including connections establishing)
tps = 2.893436 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux Desktop]$  pgbench -f Query16 -U vm -t 10000 project
starting vacuum...end.
transaction type: Query16
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 369.679 ms
tps = 2.705047 (including connections establishing)
tps = 2.705050 (excluding connections establishing)
```

Plan and Execution Time Before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * fr...	10000	2434.5610170000004	3317774.8066590168

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * fr...	10000	2797.652582000005	3564036.0411070134
2	/*pga4da...	3845	9132.820939000005	67753.07573799996

Index Scan After:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
8	16568	16594	public	src_cd	src_cd_pkey	80008	80008	80008
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_bran...	pgbench_branches_pkey	0	0	0
11	16682	16697	public	pgbench_tellers	pgbench_tellers_pkey	0	0	0
12	16685	16699	public	pgbench_acco...	pgbench_accounts_pkey	0	0	0
13	16556	32802	public	nut_data	idx_food_des_ndb_no_hash	0	0	0
14	16556	32803	public	nut_data	idx_nut_data_ndb_no_hash	0	0	0
15	16556	32804	public	nut_data	idx_nut_data_ndb_no	20004	20004	0
16	16556	32805	public	nut_data	idx_nut_data_src_cd	0	0	0
17	16556	32806	public	nut_data	idx_nut_data_src_cd_hash	0	0	0
18	16568	32807	public	src_cd	idx_src_cd_src_cd_hash	0	0	0

Query: 17

```
Select * from src_cd sc Inner join nut_data nd On sc.src_cd = nd.src_cd
Inner join food_des fd On fd.ndb_no = nd. ndb_no
Where sc.src_cd = 10;
```

Attribute:

B-tree on `ndb_no` in `nut_data` table
Hash on `nd.src_cd` in `nut_data` table

Most expedient index:

USED: `CREATE INDEX idx_nut_data_ndb_no ON nut_data (ndb_no);`
NOT USED: `CREATE INDEX idx_nut_data_src_cd_hash ON nut_data using hash (src_cd);`

Effect of the index:

Running 10k	Before	After
Total plan time	2,424.57	2,372
Total exec time	89.92	90.52
Tps	2544.11	2462
Latency	0.394	0.407
Index scans	0	idx_nut_data_ndb_no = 20000 idx_nut_data_src_cd_hash = 0
Query Planner	https://explain.dalibo.com/plan/hgf1a91669f039g8#plan	https://explain.dalibo.com/plan/55886cce5555ba25

Justification:

- **Attribute and Index Choice:** A hash index on `src_cd` was appropriate for direct equality checks (`WHERE src_cd = 2`). Hash indexes are efficient for exact-match queries and avoid the overhead of scanning large datasets and it was selected by query planner but not scanned. B-tree index on `ndb_no` allowed faster joins between `nut_data` and `food_des`, leveraging the equality condition `(ON fd.ndb_no = nd.ndb_no)` trying to check if the single attribute will work out better.
- **Before vs After Comparison:** There is no improvement regarding the performance of the query where the tps went from **2544** to **2462** and latency from **0.394** to **0.407** with a very insignificant change resulting in no improvement.

- **Justification:**
  Already indexed Join: In the join between the three tables all of them already have their own indexes and when we try to make an index on `ndb_no` in table `nut_data` to see if a single attribute index is better than the composite there was no effect on performance despite the usage of the index.

  Zero Rows condition: Having the condition `src_cd = 10` resulting in zero rows from the table results in the rest of operations never executing because doing the join with the other side having a zero output makes it impossible to show any effect of adding indexes.

  Small Table Size `(src_cd)`:The `src_cd` table has only 10 rows, making sequential scans highly efficient. For such a small table, the cost of scanning all rows is minimal, and the planner may not choose to use an index, as it would not provide a significant benefit over scanning the entire table.
- **Result:** no significant query optimization

Latency & TPS before:

```
[vm@archlinux Desktop]$  pgbench -f Query17 -U vm -t 10000 project
starting vacuum...end.
transaction type: Query17
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 0.394 ms
tps = 2539.820704 (including connections establishing)
tps = 2544.110940 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux Desktop]$  pgbench -f Query17 -U vm -t 10000 project
starting vacuum...end.
transaction type: Query17
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 0.407 ms
tps = 2459.185178 (including connections establishing)
tps = 2462.563406 (excluding connections establishing)
```

Plan and Execution Time Before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * fr...	10000	2424.5701089999966	89.92118599999978

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * fr...	10000	2372.7574850000033	90.54550600000002

Index Scan After:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
4	16538	16586	public	fd_group	fd_group_pkey	0	0	0
5	16544	16588	public	food_des	food_des_pkey	580032	580032	0
6	16556	16590	public	nut_data	nut_data_pkey	120012	120012	0
7	16562	16592	public	nutr_def	nutr_def_pkey	0	0	0
8	16568	16594	public	src_cd	src_cd_pkey	80016	80016	80016
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_bran...	pgbench_branches_pkey	0	0	0
11	16682	16697	public	pgbench_tellers	pgbench_tellers_pkey	0	0	0
12	16685	16699	public	pgbench_acco...	pgbench_accounts_pkey	0	0	0
13	16556	32817	public	nut_data	idx_nut_data_ndb_no	20002	20002	0
14	16556	32818	public	nut_data	idx_nut_data_src_cd_hash	0	0	0

Query: 18

```
Select * from src_cd sc Inner join nut_data nd On sc.src_cd = nd.src_cd
Inner join food_des fd On fd.ndb_no = nd. ndb_no
Where nd.src_cd in (1, 4, 5, 6, 7, 8);
```

Attribute:

Btree on `ndb_no` in `nut_data` table

Most expedient index:

```
CREATE INDEX idx_nut_data_ndb_no ON nut_data (ndb_no);
```

Effect of the index:

Running 1k	Before	After
Total plan time	252.39	255.98
Total exec time	314,191.24	316,177.05
Tps	3.06	3.04
Latency	326.62	328.507
Index scans		idx_nut_data_ndb_no = 2,000
Query Planner	https://explain.dalibo.com/plan/22e8bde13dbe5a8a	https://explain.dalibo.com/plan/3fab9c425856dbb1

Justification:

- **Attribute and Index Choice:** B-tree index on `ndb_no` allowed faster joins between `nut_data` and `food_des`, leveraging the equality condition (`ON fd.ndb_no = nd.ndb_no`) to try to check if the single attribute will work better despite the fact that hash was tried but not scanned.
- **Before vs After Comparison:** There is no improvement regarding the performance of the query where the tps went from **3.06** to **3.04** and latency from **326.62** to **328.507** with a very insignificant change resulting in no improvement.
- **Justification:** Having a condition with **a very low selectivity** since we take 6 out of the possible 10 values in the table so it removed only **22,760** rows by the filter which is around **8%** of the original **231,065** rows making it similar case to the one with no condition at all. `nut_data` is large and the `src_cd` join condition isn't selective enough, hence the planner determines that scanning the entire `nut_data` table is more efficient than using an index. The composite index `(ndb_no, nutr_no)` may already provide an efficient access

path for `ndb_no`, which limits the performance gains of adding a single-column index on `ndb_no`.

- **Result:** no significant query optimization

- **Extra:** The optimizer relies on table statistics to estimate how many rows match the filter. If the statistics indicate that a filter matches a large portion of the data, the planner will favor strategies like `Hash Joins` and `Seq Scans`. If the statistics show that only a small number of rows match, the planner may use `Index Scans` and `Nested Loops`. We have here an estimation of 6 values out of the `V(Src_cd, src_cd)` which are 10, making the statistics show we are getting a large portion so doing hash scan in it will significantly improve

Latency & TPS before:

```
[vm@archlinux Desktop]$  pgbench -f Query18 -U vm -t 1000 project
starting vacuum...end.
transaction type: Query18
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 326.627 ms
tps = 3.061598 (including connections establishing)
tps = 3.061619 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux Desktop]$  pgbench -f Query18 -U vm -t 1000 project
starting vacuum...end.
transaction type: Query18
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 1000/1000
latency average = 328.507 ms
tps = 3.044072 (including connections establishing)
tps = 3.044097 (excluding connections establishing)
```

Plan and Execution Time Before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * fr...	1000	252.3906729999998	314191.2483780003

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select * fr...	1000	255.98879999999977	316177.0552929999

Index Scan After:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
3	16532	16584	public	deriv_cd	deriv_cd_pkey	0	0	0
4	16538	16586	public	fd_group	fd_group_pkey	0	0	0
5	16544	16588	public	food_des	food_des_pkey	4000	4000	0
6	16556	16590	public	nut_data	nut_data_pkey	2000	2000	0
7	16562	16592	public	nutr_def	nutr_def_pkey	0	0	0
8	16568	16594	public	src_cd	src_cd_pkey	8000	8000	8000
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_br...	pgbench_branches_pkey	0	0	0
11	16682	16697	public	pgbench_tel...	pgbench_tellers_pkey	0	0	0
12	16685	16699	public	pgbench_ac...	pgbench_accounts_pkey	0	0	0
13	16556	32852	public	nut_data	idx_nut_data_ndb_no	2000	2000	0

Points regarding Queries from 15 to 18

Both Query15 and Query17 have the same case of filtering with a condition resulting in zero rows causing a never executed join leaving no room for optimization beside the small table of `src_cd` that don't need index to optimize the condition as its just 10 rows, Regarding Query16 and Query 18 both have a very low selectivity where one has no condition and the other has a low selectivity condition making it like no condition at all so in both no room for indexes whether in join or low selective condition

Query: 19

```
Select ndb_no, num_studies, nd.deriv_cd From deriv_cd dc
Inner join nut_data nd On dc.deriv_cd = nd.deriv_cd
Where dc.derivcd_desc like '%food%'
Order by nd.deriv_cd;
```

Attribute:

On `deriv_cd` in table `nut_data`

Most expedient index:

```
CREATE INDEX idx_nut_data_composite ON nut_data (deriv_cd,
num_studies, ndb_no);
```

Effect of the index:

Running 10k	Before	After
Total plan time	1653.63	1634.352
Total exec time	526,999.89	122,488.54
Tps	18.44	76.27
Latency	54.204	13.11
Index scans		idx_nut_data_composite = 230,000
Query Planner	https://explain.dalibo.com/plan/81255812bce119fd	https://explain.dalibo.com/plan/9e0effe4ga7494b9

Justification:

- **Attribute and index choice:** We do joining with the attribute `deriv_cd` where it's indexed in a table and in `nut_data` not so I used it in order to facilitate the join making it a targeted attribute for index. A BTREE index was implemented as it efficiently supports equality-based lookups and range queries. Given that `deriv_cd` is commonly used for equality checks in joins and filters, a BTREE index was the most suitable type, ensuring rapid access and optimal query performance.
- **Before vs After Comparison:** The index was scanned as seen in the index scans but no improvement happened regarding tps/latency and execution time. Tps increased from **18.44** to **76.27** by **57.83 ms** and latency decreased from **54.204** to **13.11** by **41.09 ms**, which is **so insignificant**. The actual execution time decreased from **526,999** to **122,488 ms**, a reduction of about **76%** compared to the previous execution time **of 482,300 ms**.
- **Justification:** Choosing to optimize on the composite of the projection attributes beside the selection of the join attribute result in **reduced Data Access in** `nut_data` **Table**: The index

allows direct access to the relevant columns (`deriv_cd, num_studies, ndb_no`) and ensures that only rows matching the `deriv_cd` in `nut_data` are fetched efficiently. This reduces the time needed to identify matching rows because the search is constrained to just what is necessary, resulting in faster matching of the 71536 rows instead of 253,825 then having a join of cost :

$$T(W) = \frac{T(filtered\ deriv\_cd) * T(nut\_data)}{\max\left(V(\ deriv\_cd, A), V(nut\_data, A)\right)} = \frac{253{,}825 * 21}{\max\left(54, 54\right)} = \frac{5{,}330{,}325}{54} = 98{,}709.72$$

$$T(W) = \frac{T(filtered\ deriv\_cd) * T(filtered\ nut\_data)}{\max\left(V(\ deriv\_cd, A), V(nut\_data, A)\right)} = \frac{71536 * 21}{\max\left(54, 54\right)} = \frac{1502256}{54} = 27{,}819.55$$

- **The following are insights on the other possible indexes:**
    - Leading Wildcard in `LIKE` Condition: The condition `dc.derivcd_desc LIKE '%food%'` uses a leading wildcard, preventing the use of standard BTREE or hash indexes forces the planner to **check every row of the 54** in `deriv_cd` rather than jumping directly to potentially matching rows using an index.
    - Low selectivity and small table: Having the condition `dc.derivcd_desc LIKE '%food%'` only to **filter a 21 rows** out of a very **small table of 54 rows**
    - Sorting with external merge: Where unlike the next query the sorting is operating on **71,536 rows** making it a costly sort and also no index can optimize it, as it already needs to pass by all chunks in the relation.
    - Heavy join: Where we are **joining 253,825 with 21 rows** which contributes to around **40%** of the time and it already have index on their attributes
- **Result:** significant query optimization

Latency & TPS before:

```
[vm@archlinux Desktop]$  pgbench -f Query19 -U vm -t 10000 project
starting vacuum...end.

transaction type: Query19
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 54.204 ms
tps = 18.448830 (including connections establishing)
tps = 18.448984 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux Desktop]$  pgbench -f Query19 -U vm -t 10000 project
starting vacuum...end.

transaction type: Query19
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 13.110 ms
tps = 76.274969 (including connections establishing)
tps = 76.276508 (excluding connections establishing)
```

Plan and Execution Time Before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select nd...	10000	1653.6396709999972	526999.8977169998

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select nd...	10000	1634.3527920000017	122488.547901

Index Scan After:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
3	16532	16584	public	deriv_cd	deriv_cd_pkey	125507	1636014	1636014
4	16538	16586	public	fd_group	fd_group_pkey	0	0	0
5	16544	16588	public	food_des	food_des_pkey	4000	4000	0
6	16556	16590	public	nut_data	nut_data_pkey	2000	2000	0
7	16562	16592	public	nutr_def	nutr_def_pkey	0	0	0
8	16568	16594	public	src_cd	src_cd_pkey	8000	8000	8000
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_bran...	pgbench_branches_pkey	0	0	0
11	16682	16697	public	pgbench_tellers	pgbench_tellers_pkey	0	0	0
12	16685	16699	public	pgbench_acco...	pgbench_accounts_pkey	0	0	0
13	16556	32864	public	nut_data	q19	230000	715380000	0

Query: 20

```
Select ndb_no, num_studies, nd.deriv_cd
From deriv_cd dc inner join nut_data nd On dc.deriv_cd = nd.deriv_cd
Where dc.derivcd_desc like '%food%' And nd.deriv_cd = 'BFYN'
Order by nd.deriv_cd
```

Attribute:

On `deriv_cd` in table `nut_data`

Most expedient index:

```
CREATE INDEX idx_nut_data_deriv_cd ON nut_data (deriv_cd);
```

Effect of the index:

Running 10k	Before	After
Total plan time	1,160.08	442.65
Total exec time	122,841	2,707.28
Tps	78.65	2513
Latency	12.715	0.398
Index scans		idx_nut_data_deriv_cd = 10,000
Query Planner	https://explain.dalibo.com/plan/6cf15d88e9508c57#plan	https://explain.dalibo.com/plan/de9f9c04cd288765#plan

Justification:

- **Attribute and index choice:** We do the join with the attribute `deriv_cd` where it's indexed in its table but not in `nut_data` so I used it in order to facilitate the join and for the use of this attribute in filtering with `nd.deriv_cd='BFYN'` making it a targeted attribute for index. BTREE index was implemented as it efficiently supports equality-based lookups and range queries. Given that `deriv_cd` is commonly used for equality checks in joins and filters, a BTREE index was the most suitable type, ensuring rapid access and optimal query performance.
- **Before vs After Comparison:** The index was used as seen in the index scans with a significant improvement regarding tps/latency and execution time. Tps increased by **2,494.35** around **99.25%** and latency decreased by **12.31 ms** around **96.87%**. The actual execution time decreased to **2,707 ms**, a reduction of about **97.8%** from the previous **122,841 ms**, showcasing the index's impact.

- **Justification:** The condition `(deriv_cd = 'BFYN')` scanned **126,540 rows** and retained only **746 matching rows**, removing around **99%** of the rows indicating it's a **very selective** condition resulting in an efficient use of index. After creating the index `idx_nut_data_deriv_cd` on `nut_data.deriv_cd`, the performance improved drastically. The query planner adopted a Bitmap Index Scan (using bitmap index on-the-fly), significantly reducing scan times and avoiding full table scans. The sort happens after the join step when the final result set is prepared, but it does not affect the performance because it is small enough with **only** 746 to be done in-memory without significant cost unlike the previous query where the output rows were **71,536** before sorting them.
- **Result:** Significant query optimization

Latency & TPS before:

```
[vm@archlinux Desktop]$  pgbench -f Query20  -U vm -t 10000 project
starting vacuum...end.
transaction type: Query20
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 12.715 ms
tps = 78.647421 (including connections establishing)
tps = 78.650042 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux Desktop]$  pgbench -f Query20  -U vm -t 10000 project
starting vacuum...end.
transaction type: Query20
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 0.398 ms
tps = 2511.574159 (including connections establishing)
tps = 2513.104992 (excluding connections establishing)
```

Plan and Execution Time Before:

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select ndb_no, num_studies, nd.deriv_cd f...	20000	3328.5845849999987	970835.1672950014
2	select ndb_no, num_studies, nd.deriv_cd f...	10000	1160.084604999993	122841.82774400017

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select ndb_no, num_studies, nd...	10000	442.65995499999934	2707.282203000003

Index Scan After:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
4	16538	16586	public	fd_group	fd_group_pkey	0	0	0
5	16544	16588	public	food_des	food_des_pkey	20002	20002	0
6	16556	16590	public	nut_data	nut_data_pkey	20002	20002	0
7	16562	16592	public	nutr_def	nutr_def_pkey	0	0	0
8	16568	16594	public	src_cd	src_cd_pkey	0	0	0
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_bran...	pgbench_branches_pkey	0	0	0
11	16682	16697	public	pgbench_tellers	pgbench_tellers_pkey	0	0	0
12	16685	16699	public	pgbench_acco...	pgbench_accounts_pkey	0	0	0
13	16556	24599	public	nut_data	idx_nut_data_deriv_cd_hash	0	0	0
14	16532	24600	public	deriv_cd	newidx_nut_data_deriv_cd_bfyn	0	0	0
15	16532	24601	public	deriv_cd	idx_deriv_cd_deriv_cd_hash	0	0	0
16	16556	24602	public	nut_data	idx_nut_data_deriv_cd	10000	7460000	0

Query: 21

```
Select ndb_no, num_studies, nd.deriv_cd From deriv_cd dc
Inner join nut_data nd On dc.deriv_cd = nd.deriv_cd
Where dc.derivcd_desc like '%food%' And nd.deriv_cd = 'BFYN';
```

Attribute:

On `deriv_cd` in table `nut_data`

Most expedient index:

```
CREATE INDEX idx_nut_data_deriv_cd ON nut_data (deriv_cd);
```

Effect of the index:

Running 10k	Before	After
Total plan time	1,106.29	601.36
Total exec time	120,816	3,006.98
Tps	80.11	1928.09
Latency	12.482	0.519
Index scans		idx_nut_data_deriv_cd = 10,000
Query Planner	https://explain.dalibo.com/plan/3d2477dgdd5c1131	https://explain.dalibo.com/plan/27bfeb9ah8976ca2#plan

Justification:

- **Attribute and index choice:** **Note:** the planner before and after is exactly like the previous one indicating that order by wasn't the case of improvement We do joining with the attribute `deriv_cd` where it's indexed in a table but not in `nut_data` so I used it in order to facilitate the join and for the use of this attribute filtering `nd.deriv_cd = 'BFYN'` making it a targeted attribute for index. BTREE index was implemented as it efficiently supports equality-based lookups and range queries. Given that `deriv_cd` is commonly used for equality checks in joins and filters, a BTREE index was the most suitable type, ensuring rapid access and optimal query performance.
- **Before vs After Comparison:** The index was used as seen in the index scans with a significant improvement regarding tps/latency and execution time. Tps increased by **1847.98** (around **95.86%)** and latency decreased by **11.963 ms** (around **95.85%).** The execution time was reduced from **120,816 ms** to **3,006 ms**, showcasing the index's impact.

- **Justification:** The condition `(deriv_cd = 'BFYN')` scanned **126,540 rows** and retained **only 746** matching rows, hence around **99%** were removed indicating it's a very selective condition resulting in an efficient index. After creating the index `idx_nut_data_deriv_cd` on `nut_data.deriv_cd`, the performance improved drastically. The query planner adopted a Bitmap Index Scan, significantly reducing scan times and avoiding full table scans.
- **Result:** significant query optimization

Latency & TPS before:

```
[vm@archlinux Desktop]$  pgbench -f Query21  -U vm -t 10000 project
starting vacuum...end.
transaction type: Query21
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 12.482 ms
tps = 80.116347 (including connections establishing)
tps = 80.117524 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux Desktop]$  pgbench -f Query21  -U vm -t 10000 project
starting vacuum...end.
transaction type: Query21
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 0.519 ms
tps = 1926.782927 (including connections establishing)
tps = 1928.099526 (excluding connections establishing)
```

Plan and Execution Time before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select ndb_no, num_studies, nd.d...	10000	1106.295172999992	120816.129366

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select nd...	10000	601.3634119999992	3006.9809330000116

Index Scan After:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
2	16529	16582	public	datsrcln	datsrcln_pkey	0	0	0
3	16532	16584	public	deriv_cd	deriv_cd_pkey	60028	60028	60028
4	16538	16586	public	fd_group	fd_group_pkey	0	0	0
5	16544	16588	public	food_des	food_des_pkey	20002	20002	0
6	16556	16590	public	nut_data	nut_data_pkey	20002	20002	0
7	16562	16592	public	nutr_def	nutr_def_pkey	0	0	0
8	16568	16594	public	src_cd	src_cd_pkey	0	0	0
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_br...	pgbench_branches...	0	0	0
11	16682	16697	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
12	16685	16699	public	pgbench_ac...	pgbench_account...	0	0	0
13	16556	24609	public	nut_data	idx_nut_data_deriv...	10000	7460000	0

Query: 22

```
SELECT ndb_no, num_studies, nd.deriv_cd
FROM deriv_cd dc inner join nut_data nd On dc.deriv_cd = nd.deriv_cd
Where dc.derivcd_desc like '%food%' And nd.deriv_cd = 'AI';
```

Attribute:

On `deriv_cd` in table `nut_data`

Most expedient index:

```
CREATE INDEX idx_nut_data_deriv_cd ON nut_data (deriv_cd);
```

Effect of the index:

Running 10k	Before	After
Total plan time	554.47	520.82
Total exec time	121.23	120.82
Tps	2561.36	2512.93
Latency	0.391	0.398
Index scans	None	None
Query Planner	https://explain.dalibo.com/plan/f142ad575c740556	https://explain.dalibo.com/plan/4b709ee39fe848hf

Justification:

- **Attribute and index choice:** We are joining with the attribute `deriv_cd` where it's indexed in a table and not in `nut_data` so I used it in order to facilitate the join and for the use of this attribute in filtering `nd.deriv_cd = 'AI'` making it a targeted attribute for index. A BTREE index was implemented as it efficiently supports equality-based lookups and range queries. Given that `deriv_cd` is commonly used for equality checks in joins and filters, a BTREE index was the most suitable type, ensuring rapid access and optimal query performance.
- **Before vs After Comparison:** The index was not used as seen in the index scans so no improvement regarding tps/latency and execution time. Tps decreased by **49** and latency increased by **0.007 ms**, which is insignificant change.
- **Justification:**
    - Leading Wildcard in `LIKE` Condition: The condition `dc.derivcd_desc LIKE '%food%'` uses a leading wildcard, preventing the use of standard BTREE or hash

indexes forces the planner to check every row of the 54 in `deriv_cd` rather than jumping directly to potentially matching rows using an index, also the filter condition has low selectivity

- No Matches for `deriv_cd = 'AI'`: The query includes the filter `nd.deriv_cd = 'AI'`, but no data exists matching this condition. Where the filtering of 54 rows resulted in 0.
- Execution Implication: The planner prepares for an index scan on `nut_data (idx_nut_data_deriv_cd_hash),` but it is not executed because the join condition from `deriv_cd` already results in zero matching rows. The same reason goes for why `CREATE INDEX idx_nut_data_composite ON nut_data (deriv_cd, num_studies, ndb_no)` will not beneficial.

- **Result:** no query optimization

Latency & TPS before:

```
[vm@archlinux Desktop]$  pgbench -f Query22 -U vm -t 10000 project
starting vacuum...end.
transaction type: Query22
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 0.391 ms
tps = 2559.690034 (including connections establishing)
tps = 2561.364338 (excluding connections establishing)
```

Latency & TPS After:

```
[vm@archlinux Desktop]$  pgbench -f Query22 -U vm -t 10000 project
starting vacuum...end.
transaction type: Query22
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10000
number of transactions actually processed: 10000/10000
latency average = 0.398 ms
tps = 2511.324395 (including connections establishing)
tps = 2512.939756 (excluding connections establishing)
```

Plan and Execution Time before:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select nd...	10000	554.4767539999991	121.2334720000005

Plan and Execution Time After:

Data Output | Explain | Messages | Notifications

	query text	calls bigint	total_plan_time double precision	total_exec_time double precision
1	select nd...	10000	520.8284549999971	120.82452900000004

Index Scan Before and After:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
3	16532	16584	public	deriv_cd	deriv_cd_pkey	0	0	0
4	16538	16586	public	fd_group	fd_group_pkey	0	0	0
5	16544	16588	public	food_des	food_des_pkey	0	0	0
6	16556	16590	public	nut_data	nut_data_pkey	0	0	0
7	16562	16592	public	nutr_def	nutr_def_pkey	0	0	0
8	16568	16594	public	src_cd	src_cd_pkey	0	0	0
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_bran...	pgbench_branches_pkey	0	0	0
11	16682	16697	public	pgbench_tellers	pgbench_tellers_pkey	0	0	0
12	16685	16699	public	pgbench_acco...	pgbench_accounts_pkey	0	0	0
13	16556	32836	public	nut_data	idx_nut_data_deriv_cd	0	0	0

I used the following and they didn't optimize the query:

Data Output | Explain | Messages | Notifications

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
2	16529	16582	public	datsrcln	datsrcln_pkey	27176	547653173	93845
3	16532	16584	public	deriv_cd	deriv_cd_pkey	8002	8002	8002
4	16538	16586	public	fd_group	fd_group_pkey	0	0	0
5	16544	16588	public	food_des	food_des_pkey	1640744	1640744	0
6	16556	16590	public	nut_data	nut_data_pkey	345118006	345371671	343719257
7	16562	16592	public	nutr_def	nutr_def_pkey	0	0	0
8	16568	16594	public	src_cd	src_cd_pkey	47588	47588	47588
9	16574	16596	public	weight	weight_pkey	0	0	0
10	16688	16695	public	pgbench_br...	pgbench_branches...	0	0	0
11	16682	16697	public	pgbench_tel...	pgbench_tellers_p...	0	0	0
12	16685	16699	public	pgbench_ac...	pgbench_account...	0	0	0
13	16532	16846	public	deriv_cd	derivcd_desc_dc_b...	0	0	0
14	16532	16847	public	deriv_cd	derivcd_desc_dc_h...	0	0	0
15	16556	16850	public	nut_data	deriv_cd_nd_btree	0	0	0
16	16556	16851	public	nut_data	deriv_cd_nd_hash	0	0	0

Points regarding Queries from 19 to 22:

In query 19 we have a low selectivity query and costly join between 3 tables causing low performance with space for optimization on nut_data to filter before the join but once we go to high selectivity in query 20 we start to have a space for improvement so the index improves the performance of the query. Query 21 revealed to us that order by wasn't the object of optimization because with it or without it doesn't matter. Finally, query 22 states that when there is no rows after the selection the rest of the joining isn't executed so the index won't improve it whether it will be used or not.

Hash Join in the Query19: The database chooses a hash join for the first query because the condition `derivcd_desc like '%food%'` does not reduce the search space enough to justify using an index scan. A hash join works well when large sets of data need to be joined and the index does not provide a significant advantage.

Bitmap and Parallel Scans in the Second Query 20: When the `deriv_cd = 'BFYN'` condition is added, the optimizer realizes that it can reduce the table more effectively before performing the join. This leads to a bitmap index scan, which can more efficiently for combining multiple conditions and minimizing the rows scanned. Parallel scans on `nut_data` allow better utilization of system resources and reduce the I/O wait time.