Database Programming Project Milestone 2 Report

Team Members:

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

10.12.2024

Sarah El-Feel
Queries [1-8]

Query: 1

```
select * from food_des fd inner join fd_group fg on fd.fdgrp_cd = fg.fdgrp_cd
where fg.fddrp_desc = 'Breakfast Cereals';
```

Attribute:

Index on `(fdgrp_cd)` in `food_des`

Most expedient index:

```
CREATE INDEX idx_fooddes_fdgrp_cd ON food_des(fdgrp_cd);
```

Effect of the index:

	Before	**After**
Planning time	0.199 ms	0.212 ms
Execution time	1.431 ms	0.146 ms
Highest cost	Seq Scan on food_des = 233.46 rows	Index Scan on food_des = 162
Slowest runtime	Hash Join on two tables = 0.717 ms	Nested loop = 0.059 ms
Largest number of rows	Seq Scan on food_des = 7146 rows	Index Scan on food_des = 403 Nested loop on food_des = 403
Query Planner	https://explain.dalibo.com/plan/14g2g324f3efc6cg	https://explain.dalibo.com/plan/h5b69ea691d0ef5f

Plan before:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Hash Join (cost=1.31..256.85 rows=298 width=175) (actual time=0.406..1.349 rows=403 loops=1)
2	Hash Cond: (fd.fdgrp_cd = fg.fdgrp_cd)
3	-> Seq Scan on food_des fd (cost=0.00..233.46 rows=7146 width=151) (actual time=0.013..0.622 rows=7146 loops=1)
4	-> Hash (cost=1.30..1.30 rows=1 width=24) (actual time=0.009..0.010 rows=1 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Seq Scan on fd_group fg (cost=0.00..1.30 rows=1 width=24) (actual time=0.005..0.007 rows=1 loops=1)
7	Filter: (fddrp_desc = 'Breakfast Cereals'::text)
8	Rows Removed by Filter: 23
9	Planning Time: 0.199 ms
10	Execution Time: 1.431 ms

Plan after:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.28..166.13 rows=298 width=175) (actual time=0.017..0.118 rows=403 loops=1)
2	-> Seq Scan on fd_group fg (cost=0.00..1.30 rows=1 width=24) (actual time=0.006..0.008 rows=1 loops=1)
3	Filter: (fddrp_desc = 'Breakfast Cereals'::text)
4	Rows Removed by Filter: 23
5	-> Index Scan using idx_fooddes_fdgrp_cd on food_des fd (cost=0.28..161.85 rows=298 width=151) (actual time=0.008..0.051 rows=403 loops=1)
6	Index Cond: (fdgrp_cd = fg.fdgrp_cd)
7	Planning Time: 0.212 ms
8	Execution Time: 0.146 ms

Justification:

Justification for attribute: Since in our table we are joining two tables and then filtering by `fddrp_desc`, it is best that there is an index on the join attribute `fdgrp_cd` in `food_des` such that instead of joining 7146, it gets reduced to a lower number. An index on the filtering attribute is not efficient since the table is only 24 rows and the overhead of creating the index is greater than the benefits we would get out of creating the index.

Justification for index: A B+Tree Index is good for range and exact matches, hence I created a B+Tree index that can find the relevant rows we need for the join. Postgres usually preferers B+Tree indexes for joins.

Justification for observed behavior:

Sequential Scan (Before) vs. Index Scan (After) on `food_des`

- **Before:** The query scanned all **7146** rows in `food_des`, regardless of whether they matched `fdgrp_cd`. This was resource-intensive due to the table's size. The cost of passing through the whole table is high since we would have to scan the whole table which means an I/O per row.
- **After:** The index allowed the query to directly access only the **403** matching rows in `food_des`. This reduced the scan cost and runtime significantly. It also reduced the number of rows that will be joined with `fd_group`. The cost reduced, suggesting that selecting the only needed rows results in a lower cost.

Hash Join (Before) vs. Nested Loop Join (After)

- **Before:** A hash join was necessary because both tables were sequentially scanned. The join required building and searching a hash table, increasing cost and runtime. The highest runtime is that of the hash join since you are considering all rows.
- **After:** The index on `fdgrp_cd` enabled efficient row-by-row matching, making the nested loop join optimal for the query. This avoided the overhead of hashing. The

runtime of the join reduced significantly after the nested loop join was used since we are only joining the needed rows without having any unnecessary joins.

Reduced Execution Time

- Execution time improved from `1.431 ms` to `0.146 ms` due to:
    - Fewer rows scanned in `food_des` (7146 → 403).
    - Elimination of the hash table overhead.
    - Faster nested loop join enabled by the index.

Result: Highly significant query optimization

Query: 2

```
select * from weight where seq = '1';
```

Attribute:

Index on `(seq)` in `weight`

Most expedient index:

```
CREATE INDEX idx_weight_seq ON weight(seq);
```

Effect of the index:

	Before	**After**
Planning time	0.075 ms	0.082 ms
Execution time	1.591 ms	1.215 ms
Highest cost	Seq Scan on weight = 287.61 rows	Bitmap heap scan on weight = 209
Slowest runtime	Seq Scan on weight = 1.318	Bitmap heap scan on weight =0.778 ms
Largest number of rows	Seq Scan on weight = 6666	Bitmap Index Scan on weight = 6666
Query Planner	https://explain.dalibo.com/plan/ac93404898194e15	https://explain.dalibo.com/plan/4491539a2874ahaf

Plan before:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Seq Scan on weight (cost=0.00..287.61 rows=6666 width=50) (actual time=0.011..1.318 rows=6666 loops=1)
2	Filter: (seq = '1'::bpchar)
3	Rows Removed by Filter: 6343
4	Planning Time: 0.075 ms
5	Execution Time: 1.591 ms

Plan after:

```
10  set enable_seqscan = off
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on weight (cost=79.95..288.27 rows=6666 width=50) (actual time=0.148..0.904 rows=6666 loops=1)
2	Recheck Cond: (seq = '1'::bpchar)
3	Heap Blocks: exact=124
4	-> Bitmap Index Scan on idx_weight_seq (cost=0.00..78.28 rows=6666 width=0) (actual time=0.126..0.126 rows=6666 loops=1)
5	Index Cond: (seq = '1'::bpchar)
6	Planning Time: 0.082 ms
7	Execution Time: 1.215 ms

Justification:

Justification for attribute: This query selects rows from weight that have `seq=1`. An attribute on `seq` would make it faster to find these rows without having to go through the whole table.

Justification for index: A B+Tree Index is good for range and exact matches. The reason I opted for this instead of a hash is because I was sure that using the B+Tree index I created, a bitmap index would be created once I run the query. Hence, that is why I chose B+Tree.

Disabled techniques: `set enable_seqscan = off`

Justification for observed behavior:

Sequential Scan (Before) vs. Bitmap Index Scan (After) on `seq`

- **Before:** The query scanned all `13009` rows in weight, regardless of whether they matched `seq`. This was resource-intensive due to the table's size.
- **After:** The index allowed the query to create a bitmap index on the fly to access only the `6666` matching for `seq=1`. This reduced the runtime significantly. However, even if it seems to be that the bitmap index improved the cost from **287(seq scan) to 209 (bitmap heap scan)**, it is important to note that overall cost of the after plan is still **287** due to the bitmap index scan (**78 cost**). The two operations have a total of **287**, making both plans equal in terms of cost. Creating the bitmap index on the fly in itself has a cost and that is why it seems that the cost doesn't differ. However, execution time does differ since instead of going row by row, we now use the heap scan that utilizes the bitmap index we created to fetch the needed records only.
- **Important note:** I had to enable seq scan off since after creating the index it still wanted to use the seq scan. It is reasonable that the query planner would go for a seq scan in all

cases since this is a low-selectivity query. We are returning 50% of the table, so an index would only usually cause overhead than optimization. In my opinion, if the planner did not use it from the start and was forced to use it when I turned seq_scan off, then the improvement in time is not that major. When comparing that improvement to other queries, it doesn't look like much of a difference since it's only a 25% deduction. There is small improvement, however the query itself wouldn't really need an index since it's low selective and the overhead of creating an index to return 50% of the table really is higher than just passing through them all.

Reduced Execution Time

- Execution time improved from `1.591 ms` to `1.215 ms` due to:
  - Fewer rows scanned in `weight`(13009 → 6666).
  - Elimination of seq scan overhead by using a bitmap heap scan

Result: moderate query optimization

Query: 3

`select * from weight where seq = '1' and amount > 3;`

Attribute:

Index on `(amount)` in `weight` with a partial condition on `seq`

Most expedient index:

`CREATE INDEX idx_weight_seq ON weight(amount) where seq = '1';`

Effect of the index:

	Before	After
Planning time	0.073 ms	0.090 ms
Execution time	0.754 ms	0.056 ms
Highest cost	Seq Scan on weight = 320	Index scan on weight = 131
Slowest runtime	Seq Scan on weight = 0.739	Index scan on weight = 0.041 ms
Largest number of rows	Seq Scan on weight = 60	Index Scan on weight = 60
Query Planner	https://explain.dalibo.com/plan/728bcd604e63751b	https://explain.dalibo.com/plan/4a1hhd7f8d236fc9

Plan before:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Seq Scan on weight (cost=0.00..320.13 rows=144 width=50) (actual time=0.012..0.739 rows=60 loops=1)
2	Filter: ((amount > '3'::double precision) AND (seq = '1'::bpchar))
3	Rows Removed by Filter: 12949
4	Planning Time: 0.073 ms
5	Execution Time: 0.754 ms

Plan after:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Index Scan using idx_weight_seq on weight (cost=0.28..131.10 rows=144 width=50) (actual time=0.015..0.041 rows=60 loops=1)
2	Index Cond: (amount > '3'::double precision)
3	Planning Time: 0.090 ms
4	Execution Time: 0.056 ms

Justification:

Justification for attribute: This query selects rows from `weight` that have `seq=1` and `amount>3`. Since we want an exact match for `seq` and a range for `amount`, an index has to be created that tries to optimize both these filters.

Justification for index: Instead of doing two indexes, I went for a different approach. I found out that `amount>3` is high-selective while `seq=1` is low-selective. So my index is mainly on `amount` to reduce the number of rows by a huge percentage, and I added the partial condition of `seq=1` such that we only select the rows that have `amount>3` while `seq=1`.

Justification for observed behavior:

Sequential Scan (Before) vs. Index Scan (After) on `seq`

- **Before:** The query scanned all **13009** rows in weight, regardless of whether they matched `seq and amount` or not. This was resource-intensive due to the table's size.
- **After:** The query used the index created to fetch the **60 rows** we need out of the **13009**. There is no need to do another index on seq since the index already handles fetching the rows that have `seq=1`. This is a high-selective condition since `amount>30` only appears **281** times out of **13009**, while `seq=1` appears **6666** out of **13009**. That makes the index on amount very efficient as it reduces the amount of rows by 98%. That way, instead of having **13009 I/Os**, we only have **60 I/Os** in general for fetching the correct rows. The cost of that is almost 3 times lower than a seq scan (from 320 to 131)
- **Extra (Comparison to query 2):** In query 2, although it looks the same, the index was not that much of a help since the condition `seq=1` only is low-selective. In that case, the planner opted to sequentially pass by each row. When a high-selective condition is applied like `amount > 3` in query 3, it makes sense that the planner chooses it since

we are returning a very small portion of the table that would benefit from the use of an index to fetch relevant records only.

Reduced Execution Time

- Execution time improved from `0.754 ms` to `0.056 ms` due to:
  - Fewer rows scanned in `weight`(13009 → 60).
  - Elimination of seq scan overhead by using an index scan to fetch relevant rows

Result: Highly significant query optimization

Query: 4

`select ndb_no from weight except select ndb_no from food_des;`

Attribute:

Index on `(ndb_no)` in `weight`

Most expedient index:

`CREATE INDEX idx_weight_ndb_no ON weight(ndb_no);`

Effect of the index:

	Before	After
Planning time	0.082 ms	0.356 ms
Execution time	8.241 ms	9.388 ms
Highest cost	Seq Scan on weight = 255	Index only scan on weight = 323
Slowest runtime	HashSetOp Except = 3.07	HashSetOp Except = 3.73
Largest number of rows	Append = 20155	Append = 20155
Query Planner	https://explain.dalibo.com/plan/663ebgagdbcddb27	https://explain.dalibo.com/plan/25e1ce062c6adc78

Plan before:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	HashSetOp Except (cost=0.00..803.27 rows=6679 width=28) (actual time=8.181..8.182 rows=0 loops=1)
2	-> Append (cost=0.00..752.89 rows=20155 width=28) (actual time=0.013..5.111 rows=20155 loops=1)
3	-> Subquery Scan on "*SELECT* 1" (cost=0.00..385.18 rows=13009 width=10) (actual time=0.012..2.381 rows=13009 loops=1)
4	-> Seq Scan on weight (cost=0.00..255.09 rows=13009 width=6) (actual time=0.010..1.270 rows=13009 loops=1)
5	-> Subquery Scan on "*SELECT* 2" (cost=0.28..266.93 rows=7146 width=10) (actual time=0.024..1.267 rows=7146 loops=1)
6	-> Index Only Scan using food_des_pkey on food_des (cost=0.28..195.47 rows=7146 width=6) (actual time=0.023..0.659 rows=7146 loops=1)
7	Heap Fetches: 0
8	Planning Time: 0.082 ms
9	Execution Time: 8.241 ms

Plan after:

```
set enable_seqscan = off
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	HashSetOp Except (cost=0.29..871.61 rows=6679 width=28) (actual time=9.332..9.334 rows=0 loops=1)
2	-> Append (cost=0.29..821.22 rows=20155 width=28) (actual time=0.022..5.600 rows=20155 loops=1)
3	-> Subquery Scan on "*SELECT* 1" (cost=0.29..453.51 rows=13009 width=10) (actual time=0.022..2.421 rows=13009 loops=1)
4	-> Index Only Scan using idx_weight_ndb_no on weight (cost=0.29..323.42 rows=13009 width=6) (actual time=0.020..1.154 rows=13009 loops=1)
5	Heap Fetches: 0
6	-> Subquery Scan on "*SELECT* 2" (cost=0.28..266.93 rows=7146 width=10) (actual time=0.016..1.418 rows=7146 loops=1)
7	-> Index Only Scan using food_des_pkey on food_des (cost=0.28..195.47 rows=7146 width=6) (actual time=0.015..0.726 rows=7146 loops=1)
8	Heap Fetches: 0
9	Planning Time: 0.356 ms
10	Execution Time: 9.388 ms

Justification:

Justification for attribute: This query basically wants to find the ndb_no in weight that is not in `food_des`. Since the primary key for `food_des` is `ndb_no`, there is no need to do an index on it. However, weight has a composite key of `ndb_no` (foreign key to `food_des`) and seq, so I wanted to explore if it would make a difference to create an index only on the column we're needing.

Justification for index: I created a B+Tree since it's good for search and range. All primary keys are made of B+Tree indexes so I went for the same approach.

Disabled techniques: `set enable_seqscan = off`

Justification for observed behavior:

Sequential Scan (Before) vs. Index Only Scan (After) on `ndb_no in weight`

- **Before:** The query scanned all `13009` rows in weight which is needed to be able to exclude the ones that are also in food_des.
- **After:** At first the query did not use it and went for seq scan but I forced it to use it. This resulted in index only scan on both tables. Even when I created the index, it didn't use it to fetch the values from disk but to read them only from the index table. It has a higher

cost than a normal seq scan on `weight` since the `weight` table is large and the overhead of using the index is higher than sequentially passing through the records.

Reason for no improvement:

- Logically, if we want to find `ndb_no` that is in weight and not in `food_des` it is 100% guaranteed that there will be 0 records since `ndb_no` is foreign key to `food_des` which is the primary key. If it is a foreign key, then one of its constraints is to ensure that it exists in `food_des`, which is the case here. That is why no index will be able to optimize this query since logically it does not even need an index. If I use index scan or hash scan or seq scan, they will all have to pass by the whole two tables to be able to append them and exclude them. In all cases, 0 records will be returned.

Time and cost:

- In general, cost, execution time and plan time increased since the overhead of using an index and read the values from the index are higher than just sequentially passing by the table.
- Execution time increased from `8.241 ms` to `9.388 ms` and plan time increased from `0.082 ms` to `0.356` ms since the planner has to evaluate all logical plans.

Interesting Observation:

- I realized that in `food_des`, even without using the index, the operation used was index only scan. It intrigued me, since it is logically easier to use a seq scan. But in that particular case an index only scan makes more sense since `ndb_no` in `food_des` is primary key, so instead of passing by the table, the planner used the predefined primary key index to read only, instead of fetching. That way in the comparison, there is no need to go to the disk to fetch the values. The planner can check the existing of `ndb_no` in `food_des` through the index only scan.

Result: No query optimization

Query: 5

```
select ndb_no from weight except select ndb_no from datsrcln;
```

Attribute:

Index on `(ndb_no)` in `datsrcln`

Most expedient index:

```
CREATE INDEX idx_datsrcln_ndb_no ON datsrcln(ndb_no);
```

Effect of the index:

	Before	After
Planning time	0.078 ms	0.090 ms
Execution time	53.382 ms	43.620 ms
Highest cost	Seq Scan on datsrcln = 1536	Index only scan on datsrcln = 1747
Slowest runtime	HashSetOp Except = 18.6	HashSetOp Except = 14.3
Largest number of rows	Append = 106854	Append = 106854
Query Planner	https://explain.dalibo.com/plan/07169f90693d9754	https://explain.dalibo.com/plan/4fd58ff95cb831fe

Plan before:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	HashSetOp Except (cost=0.00..3661.48 rows=6679 width=28) (actual time=52.527..53.027 rows=5077 loops=1)
2	-> Append (cost=0.00..3394.35 rows=106854 width=28) (actual time=0.012..34.430 rows=106854 loops=1)
3	-> Subquery Scan on "*SELECT* 1" (cost=0.00..385.18 rows=13009 width=10) (actual time=0.011..2.553 rows=13009 loops=1)
4	-> Seq Scan on weight (cost=0.00..255.09 rows=13009 width=6) (actual time=0.009..1.264 rows=13009 loops=1)
5	-> Subquery Scan on "*SELECT* 2" (cost=0.00..2474.90 rows=93845 width=10) (actual time=0.013..22.835 rows=93845 loops=1)
6	-> Seq Scan on datsrcln (cost=0.00..1536.45 rows=93845 width=6) (actual time=0.012..8.356 rows=93845 loops=1)
7	Planning Time: 0.078 ms
8	Execution Time: 53.382 ms

Plan after:

```
set enable_seqscan = off
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	HashSetOp Except (cost=0.29..4021.33 rows=6679 width=28) (actual time=42.671..43.255 rows=5077 loops=1)
2	-> Append (cost=0.29..3754.20 rows=106854 width=28) (actual time=0.025..28.930 rows=106854 loops=1)
3	-> Subquery Scan on "*SELECT* 1" (cost=0.29..533.51 rows=13009 width=10) (actual time=0.024..3.050 rows=13009 loops=1)
4	-> Index Only Scan using weight_pkey on weight (cost=0.29..403.42 rows=13009 width=6) (actual time=0.022..1.761 rows=13009 loops=1)
5	Heap Fetches: 0
6	-> Subquery Scan on "*SELECT* 2" (cost=0.29..2686.42 rows=93845 width=10) (actual time=0.014..16.848 rows=93845 loops=1)
7	-> Index Only Scan using idx_datsrcln_ndbo_no on datsrcln (cost=0.29..1747.97 rows=93845 width=6) (actual time=0.013..7.495 rows=93845 loops=1)
8	Heap Fetches: 0
9	Planning Time: 0.090 ms
10	Execution Time: 43.620 ms

Justification:

Justification for attribute: This query basically wants to find the `ndb_no` in weight that is not in `food_des`. Since the primary key for `food_des` is `ndb_no`, there is no need to do an index on it. However, weight has a composite key of `ndb_no` (foreign key to `food_des`) and seq, so I wanted to explore if it would make a difference to create an index only on the column we're needing.

Justification for index: I created a B+Tree since it's good for search and range. All primary keys are made of B+Tree indexes so I went for the same approach.

Disabled techniques: `set enable_seqscan = off`

Justification for observed behavior:

Sequential Scan (Before) vs. Index Scan (After) on `weight and datsrcln`

- **Before:** The query scanned all `13009` rows in weight and `93845` in `datsrcln`. A seqcan involves fetching the tables from the disk.
- **After:** After turning seqscan off, the query used the primary key index of `weight` for an index only scan and used my created index for `datsrcln`. The index only scan performs a read on the index but does not read from the table. This could explain why the execution time is less. Using an index only scan is much more efficient than seqscan since no disk I/Os are needed. However the overhead of creating an index and using it is higher than the seqscan.

Reason for no improvement

The query tries to find `ndb_no` in weight that are not in `datsrcln`. Since we will always need all `ndb_no` of both tables, whatever physical operator we use, we will have to get all the records and then remove the ones from weight that appear in `datsrcln`. For that reason, an index did not really affect the execution time much. It increased the cost of the operation since an index only scan has a higher cost than seqscan in that case. In query 4, initially `food_des` had an index only scan on it, but here since the table in the nested query is `datsrcln` and `ndb_no` is not primary key in that table, the planner used the regular seqscan to fetch all `ndb_no`.

Time and cost:

- In general, execution time (**from 53.382 ms to 43.620 ms**) decreased while plan time (**from 0.078 ms to 0.090 ms**), cost increased since the overhead of creating more plans for execution of the query and for using an index are greater than just sequentially going through the table. The execution time difference may seem significant, but if we compare it to other queries, that drop in execution time is considered insignificant. It also fluctuates a lot so any range between 40 to 60 is considered insignificant since it's only 10-20% improvement.

Result: No significant/moderate query optimization

Query: 6

```
select nutdef.nutrdesc, w.gm_wgt from nutr_def nutdef inner join nut_data nd
on nd.nutr_no = nutdef.nutr_no inner join weight w on w.ndb_no = nd.ndb_no
where nutdef.tagname = 'CA';
```

Attribute:

Index on `(tagname)` in `nutr_def`

Most expedient index:

```
create index idx_nutr_def_tagname_hash on nutr_def using HASH(tagname);
```

Effect of the index:

	Before	After
Planning time	0.900 ms	0.421 ms
Execution time	15.228 ms	14.662 ms
Highest cost	Index Only Scan nut_data = 5843	Index Only Scan nut_data = 5843
Slowest runtime	Index Only Scan nut_data = 11 ms	Index Only Scan nut_data = 8.19 ms
Largest number of rows	Seq Scan weight = 13009	Seq Scan weight = 5718
Query Planner	https://explain.dalibo.com/plan/be73cc1ce42823dg	https://explain.dalibo.com/plan/2069ee4be5g174de

Plan before:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Hash Join (cost=5890.17..6230.41 rows=3636 width=20) (actual time=11.954..14.984 rows=4560 loops=1)
2	Hash Cond: (w.ndb_no = nd.ndb_no)
3	-> Seq Scan on weight w (cost=0.00..255.09 rows=13009 width=14) (actual time=0.013..0.983 rows=13009 loops=1)
4	-> Hash (cost=5866.85..5866.85 rows=1866 width=18) (actual time=11.930..11.932 rows=2859 loops=1)
5	Buckets: 4096 (originally 2048) Batches: 1 (originally 1) Memory Usage: 172kB
6	-> Nested Loop (cost=0.42..5866.85 rows=1866 width=18) (actual time=0.026..11.321 rows=2859 loops=1)
7	-> Seq Scan on nutr_def nutdef (cost=0.00..3.70 rows=1 width=16) (actual time=0.008..0.024 rows=1 loops=1)
8	Filter: (tagname = 'CA'::text)
9	Rows Removed by Filter: 135
10	-> Index Only Scan using nut_data_pkey on nut_data nd (cost=0.42..5843.63 rows=1952 width=10) (actual time=0.014..10.988 rows=2859 loops=1)
11	Index Cond: (nutr_no = nutdef.nutr_no)
12	Heap Fetches: 0
13	Planning Time: 0.900 ms
14	Execution Time: 15.228 ms

Plan after:

```
set enable_seqscan = off;
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.71..6510.81 rows=3636 width=20) (actual time=0.019..14.501 rows=4560 loops=1)
2	-> Nested Loop (cost=0.42..5871.17 rows=1866 width=18) (actual time=0.015..8.402 rows=2859 loops=1)
3	-> Index Scan using idx_nutr_def_tagname_hash on nutr_def nutdef (cost=0.00..8.02 rows=1 width=16) (actual time=0.006..0.008 rows=1 loops=1)
4	Index Cond: (tagname = 'CA'::text)
5	-> Index Only Scan using nut_data_pkey on nut_data nd (cost=0.42..5843.63 rows=1952 width=10) (actual time=0.008..8.187 rows=2859 loops=1)
6	Index Cond: (nutr_no = nutdef.nutr_no)
7	Heap Fetches: 0
8	-> Index Scan using weight_pkey on weight w (cost=0.29..0.32 rows=2 width=14) (actual time=0.002..0.002 rows=2 loops=2859)
9	Index Cond: (ndb_no = nd.ndb_no)
10	Planning Time: 0.421 ms
11	Execution Time: 14.662 ms

Justification:

Justification for Attribute:
The `tagname` attribute in the `nutr_def` table was indexed because it is frequently used in equality-based conditions (e.g., `tagname = 'CA'`). By indexing this column, the database can quickly locate the relevant row(s) without performing a full table scan, particularly when searching for specific `tagname` values. Other attributes in the join already have an index on them.

Justification for Index:
A **HASH** index was created on the `tagname` column because HASH indexing is optimized for equality comparisons. Unlike B+Tree indexes, HASH indexes provide constant-time lookup for equality conditions, making them a logical choice for this query where `tagname` is checked for equality with `'CA'`.

Disabled techniques: `set enable_seqscan = off`

Justification for Observed Behavior:

Hash Join (Before) vs. Nested Loop (After)

- **Before**: The query used a **Hash Join** between the `nut_data` and `weight` tables and a sequential scan on the `nutr_def` table to filter for `tagname = 'CA'`. This approach involved scanning more rows from the `weight` and `nut_data` tables, increasing execution time.

- **After**: The query switched to a **Nested Loop** join due to the HASH index on `tagname`. The index allowed for faster lookup of `nutr_def` rows matching the condition. However, despite the logical improvement in accessing `nutr_def` rows, the overall execution time did not improve as expected.

Reason for Limited Improvement:

1. **Data Size and Distribution**: The number of rows filtered by `tagname = 'CA'` is very small (**1 row out of 136**), so the HASH index does not significantly impact the overall cost compared to the other operations in the query.
2. **Dominant Costs**: The join operations between the `nut_data` and `weight` tables dominate the query execution time. While the HASH index speeds up access to `nutr_def`, the subsequent operations remain expensive due to the large number of rows (e.g., **13,009 rows** in `weight` and **2,859 rows** in the filtered `nut_data`). Even when I turned off seqscan, it is still heavy because of joining **5718** with **2859 rows**. So maybe the small improvement could be attributed to the smaller weight table size. In general, hash join is less costly than nested loops, so when we forced the planner to avoid using seq scan, the join changed from hash to nested loops which is in almost all the cases more expensive than hash join.
3. **Nested Loop Behavior**: The use of a **Nested Loop** for joining `weight` and `nut_data` leads to repetitive lookups, which can be costly when many rows need to be joined. Hash Join is usually more efficient since bucketizing the values and comparing using hashes is much better cost-wise (hash cost: 108, nested loop cost: 639). So even with a larger number of rows, hash join is better.

Time and Cost:

- **Before**: Execution time was **15.228 ms**, with the query relying on a sequential scan for filtering `nutr_def`.
- **After**: Execution time slightly improved to **14.662 ms**, with the HASH index reducing the filtering cost on `nutr_def`. However, the improvement was not substantial because the join operations remained the bottleneck.

Result: no significant query optimization

Query: 7

```
select nutdef.nutrdesc, w.gm_wgt from nutr_def nutdef inner join nut_data nd
on nd.nutr_no = nutdef.nutr_no inner join weight w on w.ndb_no = nd.ndb_no
where nutdef.tagname like 'CA%';
```

Attribute:

Index on `(nutr_no)` in `nutr_def with partial condition on tagname`

Most expedient index:

```
Create index idx_nutr_def_tagname_partial on nutr_def(nutr_no) where tagname
like 'CA%';
```

Effect of the index:

	Before	After
Planning time	0.749 ms	0.513 ms
Execution time	63.940 ms	46.952 ms
Highest cost	Seq Scan nut_data = 5409	Index only scan nut_data = 7727
Slowest runtime	Hash Join nd and nutdef = 30 ms	Index only scan nut_data = 21.5
Largest number of rows	Seq Scan nut_data = 253825	Index only scan nut_data = 253825
Query Planner	https://explain.dalibo.com/plan/f813154e51ee4cb4	https://explain.dalibo.com/plan/a552dda64d6548cg

Plan before:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Hash Join (cost=6216.71..6800.00 rows=18185 width=20) (actual time=55.765..62.335 rows=28060 loops=1)
2	Hash Cond: (w.ndb_no = nd.ndb_no)
3	-> Seq Scan on weight w (cost=0.00..255.09 rows=13009 width=14) (actual time=0.009..1.058 rows=13009 loops=1)
4	-> Hash (cost=6100.06..6100.06 rows=9332 width=18) (actual time=55.738..55.740 rows=14486 loops=1)
5	Buckets: 16384 Batches: 1 Memory Usage: 851kB
6	-> Hash Join (cost=3.76..6100.06 rows=9332 width=18) (actual time=0.031..51.977 rows=14486 loops=1)
7	Hash Cond: (nd.nutr_no = nutdef.nutr_no)
8	-> Seq Scan on nut_data nd (cost=0.00..5409.25 rows=253825 width=10) (actual time=0.002..21.920 rows=253825 loops=1)
9	-> Hash (cost=3.70..3.70 rows=5 width=16) (actual time=0.023..0.025 rows=5 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Seq Scan on nutr_def nutdef (cost=0.00..3.70 rows=5 width=16) (actual time=0.009..0.020 rows=5 loops=1)
12	Filter: (tagname ~~ 'CA%'::text)
13	Rows Removed by Filter: 131
14	Planning Time: 0.749 ms
15	Execution Time: 63.940 ms

Plan after:

```
12  set enable_seqscan = off;
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Hash Join (cost=867.49..9638.74 rows=18185 width=20) (actual time=3.161..45.647 rows=28060 loops=1)
2	Hash Cond: (nd.ndb_no = w.ndb_no)
3	-> Hash Join (cost=10.70..8425.13 rows=9332 width=18) (actual time=0.023..38.638 rows=14486 loops=1)
4	Hash Cond: (nd.nutr_no = nutdef.nutr_no)
5	-> Index Only Scan using nut_data_pkey on nut_data nd (cost=0.42..7727.80 rows=253825 width=10) (actual time=0.007..21.533 rows=253825 loops=1)
6	Heap Fetches: 0
7	-> Hash (cost=10.22..10.22 rows=5 width=16) (actual time=0.010..0.012 rows=5 loops=1)
8	Buckets: 1024 Batches: 1 Memory Usage: 9kB
9	-> Bitmap Heap Scan on nutr_def nutdef (cost=8.16..10.22 rows=5 width=16) (actual time=0.006..0.008 rows=5 loops=1)
10	Recheck Cond: (tagname ~~ 'CA%'::text)
11	Heap Blocks: exact=1
12	-> Bitmap Index Scan on idx_nutr_def_tagname_partial (cost=0.00..8.16 rows=5 width=0) (actual time=0.002..0.002 rows=5 loops=1)
13	-> Hash (cost=694.17..694.17 rows=13009 width=14) (actual time=3.126..3.126 rows=13009 loops=1)
14	Buckets: 16384 Batches: 1 Memory Usage: 713kB
15	-> Index Scan using weight_pkey on weight w (cost=0.29..694.17 rows=13009 width=14) (actual time=0.004..1.654 rows=13009 loops=1)
16	Planning Time: 0.513 ms
17	Execution Time: 46.952 ms

Justification:

Justification for Attribute:

The `nutr_no` attribute in the `nutr_def` table was indexed with a partial index because it is frequently joined with the `nut_data` table on `nutr_no`. By limiting the index to rows where `tagname LIKE 'CA%'`, the database can efficiently filter relevant rows without scanning the entire table.

Justification for Index:

A **partial B+Tree index** was created on the `nutr_no` column with the condition `tagname LIKE 'CA%'`. This choice ensures that only rows matching the specified pattern are included in the index, optimizing the query's performance by reducing the number of rows scanned during the filtering process. Partial indexes are particularly useful in cases where only a subset of the table's rows is relevant to the query condition, minimizing storage and lookup overhead.

Disabled techniques: `set enable_seqscan = off`

Justification for Observed Behavior:

Hash Join (Before) vs. Hash Join with Bitmap Scan (After)

- **Before**: The query used a Hash Join between the `nut_data` and `weight` tables and a sequential scan on the `nutr_def` table to filter rows with `tagname LIKE 'CA%'`. This approach involved scanning all **136 rows** in `nutr_def` and computing the join, increasing execution time.
- **After**: The query utilized the partial B+Tree index, switching to a Bitmap Index Scan to locate rows in `nutr_def` matching `tagname LIKE 'CA%'`. This reduced the number of rows scanned in `nutr_def` to **5 rows** only and improved query efficiency by leveraging the partial index to filter rows directly and reduce the join. Even if a table of 136 usually doesn't need an index, it wouldn't hurt to be able to locate these 5 rows faster to have a more efficient join.

Reason for Improvement:

- **Data Size and Distribution**: The partial index restricted the indexed rows to only those satisfying `tagname LIKE 'CA%'`, resulting in only 5 rows being scanned instead of the full table.
- **Index Efficiency**: The Bitmap Index Scan benefited from the partial index, allowing the database to quickly locate matching rows and proceed with the join operations.
- **Join Optimization**: The reduced filtering cost for `nutr_def` cascaded into faster join processing for `nut_data` and `weight`.
- However, when comparing this improvement to other improvements I have explored before, it is not considered a very highly significant improvement since we're only talking about 25% improvement, however given that our index was able to reduce the number of rows for join, it can be considered an improving step. Even when running multiple times, values fluctuated from 40s to 60s ms. If our table scales up, the index would be a good approach to optimize.

Time and Cost:

- Execution time was **63.940 ms**, relying on a sequential scan for filtering rows in `nutr_def` and after it improved to **46.952 ms**

Result: moderate query optimization.

Query: 8

```
select count(*) from food_des where fat_factor is null;
```

Attribute:

Index on `(fat_factor)` in `food_des`

Most expedient index:

```
CREATE INDEX idx_food_des_fat_factor on food_des(fat_factor);
```

Effect of the index:

	Before	**After**
Planning time	0.052 ms	0.066 ms
Execution time	1.129 ms	0.356 ms
Highest cost	Seq Scan food_des = 233	Index only scan food_des = 233
Slowest runtime	Seq Scan food_des = 0.943 ms	Index only scan food_des = 0.219
Largest number of rows	Seq Scan food_des = 1911	Index only scan food_des= 1911
Query Planner	https://explain.dalibo.com/plan/8g68ca56abfd46cb	https://explain.dalibo.com/plan/d8ec1400af2gde7e

Plan before:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Aggregate (cost=238.24..238.25 rows=1 width=8) (actual time=1.099..1.100 rows=1 loops=1)
2	-> Seq Scan on food_des (cost=0.00..233.46 rows=1911 width=0) (actual time=0.023..0.943 rows=1911 loops=1)
3	Filter: (fat_factor IS NULL)
4	Rows Removed by Filter: 5235
5	Planning Time: 0.052 ms
6	Execution Time: 1.129 ms

Plan after:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Aggregate (cost=62.50..62.51 rows=1 width=8) (actual time=0.331..0.332 rows=1 loops=1)
2	-> Index Only Scan using idx_food_des_fat_factor on food_des (cost=0.28..57.72 rows=1911 width=0) (actual time=0.025..0.219 rows=1911 loops=1)
3	Index Cond: (fat_factor IS NULL)
4	Heap Fetches: 0
5	Planning Time: 0.066 ms
6	Execution Time: 0.356 ms

Justification:

Justification for attribute: The `fat_factor` attribute was chosen because the query specifically filters rows where `fat_factor IS NULL.` This attribute is frequently queried for null values in the `food_des` table, and creating an index on it optimizes these queries by reducing the need to scan the entire table. As null filtering is a common operation, indexing this attribute aligns with the goal of improving query performance.

Justification for index: I created a B+Tree index on the `fat_factor` column because B+Tree indexes are efficient for search operations, including equality and range queries. Since `fat_factor` is the column used in the `WHERE` clause with a filter condition `(IS NULL`), the index allows the database to quickly locate rows that match the condition without scanning the entire table.

Justification for observed behavior:

Sequential Scan (Before) vs. Index Scan (After) on `food_des`

- **Before**: The query performed a sequential scan on the `food_des` table, which required reading all **7,146 rows**. A sequential scan involves fetching rows from the disk for every record, even when many rows do not meet the `IS NULL` condition.
- **After**: The query utilized the newly created `idx_food_des_fat_factor` index to perform an index-only scan. This scan allows the database to read directly from the index without fetching data from the table, as all required information is contained within the index. The index helped get the **1911 rows** directly without passing through the whole table.

Reason for improvement

The index-only scan efficiently retrieves the rows where `fat_factor IS NULL` directly from the index, eliminating the need to read non-relevant rows from the table. This optimization reduces the total number of I/O operations. The observed reduction in cost and execution time is due to the index precisely targeting the relevant rows. The query is also high-selective, since we return `1911` out of `7146`, which is why an index would be efficient to find 1911 rows directly.

Time and cost:

- **Planning Time**: Increased slightly from `0.052 ms` to `0.066 ms` due to the added overhead of evaluating the use of the new index.

- **Execution Time**: Decreased significantly from `1.129 ms` to `0.356 ms`, as the index-only scan avoids unnecessary table reads.

Result: Significant query optimization

Abdelrahman Elnagar

Queries [9, 16]

Query: 9
`Select * from food_des where fat_factor is null;`

Attribute:
Index on `fat_factor` in `food_des`

Most expedient index:
USED: `CREATE INDEX idx_fat_factor_null_partial ON food_des (fat_factor) WHERE fat_factor IS NULL;`
USED: `CREATE INDEX idx_fat_factor_btree ON food_des (fat_factor);` (was preferred by the planner)

Effect of the index:

	Before	After
Planning time	0.070 ms	0.098 ms
Execution time	3.184 ms	0.651 ms
Highest cost	Seq Scan on food_des = 233	Bitmap Heap Scan on food_des =182
Slowest runtime	Seq Scan on food_des = 3.01 ms	Bitmap Heap Scan on food_des =0.409 ms
Largest number of rows	Seq Scan on food_des = 5235 rows	Bitmap Heap Scan on food_des =1,911 rows
Query Planner	https://explain.dalibo.com/plan/h5acfacg75334b81	https://explain.dalibo.com/plan/0e2ha3fd4deh68b1

Plan before:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Seq Scan on food_des (cost=0.00..233.46 rows=1911 width=151) (actual time=0.030..3.008 rows=1911 loops=1)
2	Filter: (fat_factor IS NULL)
3	Rows Removed by Filter: 5235
4	Planning Time: 0.070 ms
5	Execution Time: 3.184 ms

Plan after:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on food_des (cost=39.09..220.20 rows=1911 width=151) (actual time=0.115..0.500 rows=1911 loops=1)
2	Recheck Cond: (fat_factor IS NULL)
3	Heap Blocks: exact=102
4	-> Bitmap Index Scan on idx_fat_factor_btree (cost=0.00..38.61 rows=1911 width=0) (actual time=0.090..0.091 rows=1911 loops=1)
5	Index Cond: (fat_factor IS NULL)
6	Planning Time: 0.098 ms
7	Execution Time: 0.651 ms

Justification:

Index and type justification
The Btree index is optimized for wide range queries including the `IS NULL` condition and the attribute `fat_factor` is chosen for the index as it's the one we are selecting with so when it's the index the query will perform better and also taking into consideration doing a partial index to focus more on optimizing the query by taking the condition into consideration.

Justification for observed behavior:

Sequential Scan (Before) vs. Bitmap Heap Scan (After) on `food_des`

- **Before:** The query scanned all `7146` rows in `food_des`, regardless of whether they matched `fat_factor IS NULL` or not. This was resource-intensive due to the table's size.
- **After:** The index allowed the query to create the index on the `7146` rows in `food_des`, then using the B+tree index a bitmap index on-the-fly and Bitmap Heap Scan were created, making accessing the rows with the index much easier to be able to select rows with null rather than sequentially passing through them all.

Reduced Execution Time

- Execution time improved from `3.184 ms` to `0.651 ms` due to:
  - Fewer rows scanned in `food_des` (7146 → 1911).
  - Bitmap access through index is much faster than sequential access.

Result: Significant query optimization

Query: 10

```
Select min (n_factor) from food_des where fat_factor is null;
```

Attribute:

Index on `n_factor` in `food_des`

Most expedient index:

USED: `CREATE INDEX idx_partial_n_factor ON food_des (n_factor) WHERE fat_factor IS NULL;`
USED: `CREATE INDEX idx_btree_n_factor ON food_des (n_factor);`
USED: `CREATE INDEX idx_btree_fat_factor ON food_des (fat_factor)`
USED: `CREATE INDEX idx_composite_fat_n ON food_des (fat_factor, n_factor);`
USED: `CREATE INDEX idx_covering_fat_n ON food_des (fat_factor) INCLUDE (n_factor);`

Effect of the index:

	Before	**After**
Planning time	0.265 ms	0.132 ms
Execution time	2.357 ms	0.067 ms
Highest cost	Seq Scan on food_des = 233	Index Only Scan on food_des = 54.8
Slowest runtime	Seq Scan on food_des = 2.11 ms	Result = 0.04 ms Index Only scan on food_des = 0.033
Largest number of rows	Seq Scan on food_des = 1,911 rows	Index Only Scan on food_des = 1,517 row
Query Planner	https://explain.dalibo.com/plan/c1a5a1h37ced939f	https://explain.dalibo.com/plan/7gf2bg0121e779f5

Plan before:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Aggregate (cost=238.24..238.25 rows=1 width=8) (actual time=2.283..2.290 rows=1 loops=1)
2	-> Seq Scan on food_des (cost=0.00..233.46 rows=1911 width=8) (actual time=0.052..2.113 rows=1911 loops=1)
3	Filter: (fat_factor IS NULL)
4	Rows Removed by Filter: 5235
5	Planning Time: 0.265 ms
6	Execution Time: 2.357 ms

Plan after: (Partial - Index: `idx_partial_n_factor`)

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Result (cost=0.31..0.32 rows=1 width=8) (actual time=0.039..0.040 rows=1 loops=1)
2	InitPlan 1 (returns $0)
3	-> Limit (cost=0.28..0.31 rows=1 width=8) (actual time=0.034..0.035 rows=1 loops=1)
4	-> Index Only Scan using idx_partial_n_factor on food_des (cost=0.28..54.83 rows=1517 width=8) (actual time=0.033..0.033 rows=1 loops=1)
5	Index Cond: (n_factor IS NOT NULL)
6	Heap Fetches: 0
7	Planning Time: 0.132 ms
8	Execution Time: 0.067 ms

Justification:

Index and type justification:
The B-tree index is optimized for wide range queries including the `IS NULL` condition and the attribute `fat_factor` is the chosen attribute for the index as it's the one we aggregate on. After exploring other indexes, it was found that a composite index between `(fat_factor, n_factor)` worked as well since we are filtering with `fat_factor` so either using a partial or a composite index would be used by the planner. After further exploration, it was also found that when we tried the including index in which we take the values of the `n_factor` in the index table to perform Index Only Scan (instead of index scan) it was also used and worked well. But for main optimization, the partial index was used.

Sequential Scan (Before) vs. Index Only Scan (After) on `food_des`

- **Before:** The query scanned all `7146` rows in `food_des`, regardless of whether they matched `it is null` or not. This was resource-intensive due to the table's size as we only needed 27% of the table causing high selectivity hence the need for index. Then we do **Aggregate** in order to get the `min()`.
- **After:** The index allowed the query to directly access only the `1` matching row in `food_des`. This reduced the scan cost and runtime significantly where we have just accessed the index table to get the `min()` without the need to get back to the table. Aggregate is done by initplan and then limit 1 to gets the smallest result which wraps the output and ensures that the aggregate is done and output is only a single value.

Reduced Execution Time

- Execution time improved from `2.357 ms` to `0.067 ms` due to:
    - Fewer rows scanned in `food_des` (7146 → 1).
    - Elimination of the sequential overhead.
    - Faster aggregation from `0.177 ms` to `0.042 ms` since less input.
    - Index Only Scan benefits: Eliminated the need for heap accesses, resulting in reduced I/O and overall faster query execution.

Result: Significant query optimization

Query: 11
```
Select sum (n_factor) from food_des where fat_factor is null;
```

Attribute:
Index on `n_factor` in `food_des`

Most expedient index:
USED: `CREATE INDEX idx_partial_n_factor ON food_des (n_factor) WHERE fat_factor IS NULL;`
USED: `CREATE INDEX idx_btree_fat_factor ON food_des (fat_factor) (n_factor isn't used)`
USED: `CREATE INDEX idx_composite_fat_n ON food_des (fat_factor, n_factor);`
USED: `CREATE INDEX idx_covering_fat_n ON food_des (fat_factor) INCLUDE (n_factor);`

Effect of the index:

	Before	After
Planning time	0.082 ms	0.124 ms
Execution time	2.189 ms	0.492 ms
Highest cost	Seq Scan on food_des = 233	Index Only Scan on food_des = 60.9
Slowest runtime	Seq Scan on food_des = 1.92 ms	Index Only Scan on food_des = 0.302 ms
Largest number of rows	Seq Scan on food_des = 1,911 rows	Index Only Scan on food_des = 1,911 rows
Query Planner	https://explain.dalibo.com/plan/ba1eg6b236be561f	https://explain.dalibo.com/plan/4e49aa5d458111e6

Plan before:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Aggregate (cost=238.24..238.25 rows=1 width=8) (actual time=2.143..2.145 rows=1 loops=1)
2	-> Seq Scan on food_des (cost=0.00..233.46 rows=1911 width=8) (actual time=0.037..1.916 rows=1911 loops=1)
3	Filter: (fat_factor IS NULL)
4	Rows Removed by Filter: 5235
5	Planning Time: 0.082 ms
6	Execution Time: 2.189 ms

Plan after:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Aggregate (cost=65.72..65.73 rows=1 width=8) (actual time=0.459..0.460 rows=1 loops=1)
2	-> Index Only Scan using idx_partial_n_factor on food_des (cost=0.28..60.94 rows=1911 width=8) (actual time=0.033..0.302 rows=1911 loops=1)
3	Heap Fetches: 0
4	Planning Time: 0.124 ms
5	Execution Time: 0.492 ms

Justification:

Index and type justification

The B-tree index is optimized for wide range queries including the `IS NULL` condition and the attribute `fat_factor` is the chosen attribute for the index as it's the one we aggregate on. After exploring other indexes, it was found that a composite index between `(fat_factor, n_factor)` worked as well since we are filtering with `fat_factor` so either using a partial or a composite index would be used by the planner. After further exploration, it was also found that when we tried the including index in which we take the values of the `n_factor` in the index table to perform Index Only Scan (instead of index scan) it was also used and worked well. But for main optimization, the partial index was used.

Sequential Scan (Before) vs. Index Only Scan (After) on `food_des`

- **Before:** The query scanned all `7146` rows in `food_des`, regardless of whether they matched `it is null` or not. This was resource-intensive due to the table's size as we only needed 27% of the table causing high selectivity hence the need for index. Then we do **Aggregate** in order to get the `sum()`.
- **After:** The index allowed the query to only retrieve the `1,911` rows that match `it is null` instead of doing a whole table scan. So that way, only using the index table we can get the aggregate we want without accessing the main table again then doing the aggregate. The attribute is stored in the index so our calculations can be performed without any database fetches.

Reduced Execution Time

- Execution time improved from `2.189 ms` to `0.492 ms` due to:
  - Fewer rows scanned in `food_des` (7146 → 1911).
  - Elimination of the sequential overhead.
  - Faster aggregation from `0.177 ms` to `0.042 ms` since less input.
  - Index Only Scan benefits: Eliminated the need for heap accesses, resulting in reduced I/O and overall faster query execution.

Result: Significant query optimization

Query: 12
`Select sum (n_factor) from food_des where fat_factor is not null;`

Attribute:
Index on `n_factor` in `food_des`

Most expedient index:
USED: `CREATE INDEX idx_partial_n_factor ON food_des (n_factor) WHERE fat_factor IS NOT NULL;`
USED: `CREATE INDEX idx_covering_fat_n ON food_des (fat_factor) INCLUDE (n_factor);`
USED: `CREATE INDEX idx_composite_fat_n ON food_des (fat_factor, n_factor);`

Effect of the index:

	Before	After
Planning time	0.069 ms	0.068 ms
Execution time	2.013 ms	0.622 ms
Highest cost	Seq Scan on food_des = 233	Index Only Scan on food_des = 147
Slowest runtime	Seq Scan on food_des = 1.55 ms	Index Only Scan on food_des = 0.375 ms
Largest number of rows	Seq Scan on food_des = 5,235 rows	Index Only Scan on food_des = 5,235 rows
Query Planner	https://explain.dalibo.com/plan/75b42f8e8daa0ffa	https://explain.dalibo.com/plan/44be562e609e4ahh

Plan before:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Aggregate (cost=246.55..246.56 rows=1 width=8) (actual time=1.983..1.984 rows=1 loops=1)
2	-> Seq Scan on food_des (cost=0.00..233.46 rows=5235 width=8) (actual time=0.015..1.547 rows=5235 loops=1)
3	Filter: (fat_factor IS NOT NULL)
4	Rows Removed by Filter: 1911
5	Planning Time: 0.069 ms
6	Execution Time: 2.013 ms

Plan after:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Aggregate (cost=159.90..159.91 rows=1 width=8) (actual time=0.605..0.605 rows=1 loops=1)
2	-> Index Only Scan using idx_partial_n_factor on food_des (cost=0.28..146.81 rows=5235 width=8) (actual time=0.017..0.375 rows=5235 loops=1)
3	Heap Fetches: 0
4	Planning Time: 0.065 ms
5	Execution Time: 0.622 ms

Justification:

Index and type justification

The B-tree index is optimized for wide range queries including the `IS NOT NULL` condition and the attribute `fat_factor` is the chosen attribute for the index as it's the one we aggregate on. After exploring other indexes, it was found that a composite index between `(fat_factor, n_factor)` worked as well since we are filtering with `fat_factor` so either using a partial or a composite index would be used by the planner. After further exploration, it was also found that when we tried the including index in which we take the values of the `n_factor` in the index table to perform Index Only Scan (instead of index scan) it was also used and worked well. But for main optimization, the partial index was used.

Sequential Scan (Before) vs. Index Only Scan (After) on `food_des`

- **Before:** The query scanned all `7146` rows in `food_des`, regardless of whether they matched `it is not null` or not. The operator used before was a seqscan, so even if it is not needed, it had to be fetched from disk.

- **After:** The index allowed the query to only retrieve the `5,235` rows that match `it is not null` instead of the whole table. An index only scan is useful since it does not fetch data from the disk, reducing the number of I/Os. That way, we got rid of 23% of the table and were able to use the only needed rows for the aggregation directly. Even if the query is low-selective (73% is returned), switching from seqscan to index only scan is the reason why the time and cost are lower.

Reduced Execution Time

- Execution time improved from `2.013 ms` to `0.622 ms` due to:
  - Fewer rows scanned in `food_des` (7146 → 5235).
  - Elimination of the sequential overhead.
  - **Index Only Scan benefits:** Eliminated the need for heap accesses, resulting in reduced I/O and overall faster query execution.

Result: moderate query optimization

Query: 13

```
Select tagname from nutr_def nutdef inner join datsrcln dsl on dsl.nutr_no =
nutdef.nutr_no inner join data_src ds on ds.datasrc_id = dsl.datasrc_id where
ds.vol_city = 'Cincinnati';
```

Attribute:

Partial Index on `vol_city` in `data_src`.
Composite Index on (`datasrc_id, nutr_no`) in `datsrcln`.
Covering Index on (`datasrc_id, nutr_no`) with `INCLUDE (ndb_no)` in `datsrcln`.
Covering Index on (`nutr_no`) with `INCLUDE (tagname)` in `nutr_def`.

Most expedient index:

```
USED: CREATE INDEX idx_partial_vol_city ON data_src (datasrc_id) WHERE
vol_city = 'Cincinnati';
USED: CREATE INDEX idx_datasrc_nutr_no ON datsrcln (datasrc_id, nutr_no);
USED: CREATE INDEX idx_covering_datasrc_nutr_ndb ON datsrcln (datasrc_id,
nutr_no) INCLUDE (ndb_no);
USED: CREATE INDEX idx_covering_nutr_tagname ON nutr_def (nutr_no)
INCLUDE (tagname);
```

Effect of the index:

	Before	**After**
Planning time	0.302 ms	0.279 ms
Execution time	11.188 ms	0.062 ms
Highest cost	Seq Scan on datsrcln = 1,540	Index Only Scan on datsrcln = 9.56
Slowest runtime	Hash join between datsrcln and data_src = 6.7 ms	Hash of nutr_def = 0.012 ms
Largest number of rows	Seq Scan on datsrcln = 93,845 rows	Seq Scan and hash on nutr_def = 136 rows
Query Planner	https://explain.dalibo.com/plan/g2e565c67949c87f	https://explain.dalibo.com/plan/bd7331b8b636ed7h

Plan before:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Hash Join (cost=17.65..1803.98 rows=256 width=5) (actual time=0.121..11.161 rows=27 loops=1)
2	Hash Cond: (dsl.nutr_no = nutdef.nutr_no)
3	-> Hash Join (cost=12.59..1798.23 rows=256 width=4) (actual time=0.088..11.119 rows=27 loops=1)
4	Hash Cond: (dsl.datasrc_id = ds.datasrc_id)
5	-> Seq Scan on datsrcln dsl (cost=0.00..1536.45 rows=93845 width=11) (actual time=0.003..4.349 rows=93845 loops=1)
6	-> Hash (cost=12.57..12.57 rows=1 width=7) (actual time=0.065..0.067 rows=1 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 9kB
8	-> Seq Scan on data_src ds (cost=0.00..12.57 rows=1 width=7) (actual time=0.046..0.064 rows=1 loops=1)
9	Filter: (vol_city = 'Cincinnati'::text)
10	Rows Removed by Filter: 365
11	-> Hash (cost=3.36..3.36 rows=136 width=9) (actual time=0.029..0.030 rows=136 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 14kB
13	-> Seq Scan on nutr_def nutdef (cost=0.00..3.36 rows=136 width=9) (actual time=0.007..0.015 rows=136 loops=1)
14	Planning Time: 0.302 ms
15	Execution Time: 11.188 ms

Plan after:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Hash Join (cost=5.48..26.46 rows=256 width=5) (actual time=0.038..0.045 rows=27 loops=1)
2	Hash Cond: (dsl.nutr_no = nutdef.nutr_no)
3	-> Nested Loop (cost=0.42..20.71 rows=256 width=4) (actual time=0.012..0.016 rows=27 loops=1)
4	-> Index Only Scan using idx_partial_vol_city on data_src ds (cost=0.12..8.14 rows=1 width=7) (actual time=0.003..0.003 rows=1 loops=1)
5	Heap Fetches: 1
6	-> Index Only Scan using idx_datasrc_nutr_no on datsrcln dsl (cost=0.29..9.56 rows=301 width=11) (actual time=0.007..0.008 rows=27 loops=1)
7	Index Cond: (datasrc_id = ds.datasrc_id)
8	Heap Fetches: 0
9	-> Hash (cost=3.36..3.36 rows=136 width=9) (actual time=0.023..0.023 rows=136 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 14kB
11	-> Seq Scan on nutr_def nutdef (cost=0.00..3.36 rows=136 width=9) (actual time=0.004..0.011 rows=136 loops=1)
12	Planning Time: 0.279 ms
13	Execution Time: 0.062 ms

Proof of use:

10	40983	49165	public	data_src	idx_partial_vol_city	1	1	1
11	40989	49166	public	datsrcln	idx_datasrc_nutr_no	1	27	0
12	40989	49167	public	datsrcln	idx_covering_data...	2	2	0
13	41022	49168	public	nutr_def	idx_covering_nutr_...	4	4	4

Justification:

Index and type justification

- `Partial Index (idx_partial_vol_city) on data_src (datasrc_id) WHERE vol_city = 'Cincinnati'`: Optimized for filtering rows based on the specific condition `vol_city = 'Cincinnati'`. Greatly reduces the rows scanned by directly targeting only the relevant subset of `data_src` rows, minimizing the filtering overhead.
- `Composite Index (idx_datasrc_nutr_no) on datsrcln (datasrc_id, nutr_no)`: Helps in efficiently joining `datsrcln` and `data_src` using `datasrc_id` while also supporting the `nutr_no` attribute for the subsequent join. It aligns perfectly with the query's structure, where filtering by `datasrc_id` precedes the join on `nutr_no`.
- `Covering Index (idx_covering_datasrc_nutr_ndb) on datsrcln (datasrc_id, nutr_no) INCLUDE (ndb_no):` Used for fetching `ndb_no` directly from the index without requiring access to the base table. Supports efficient execution by enabling an `Index Only Scan on datsrcln`.
- `Covering Index (idx_covering_nutr_tagname) on nutr_def (nutr_no) INCLUDE (tagname):` Allows the query to fetch `tagname` directly from the index during the join with `nutr_no`. Reduces heap access and enables an `Index Only Scan` for better performance.

Sequential Scan (Before) vs. Optimized Indexing (After)

- **Before: Sequential Scans and Hash Joins**
  Sequential scans on `data_src` and `datsrcln` resulted in scanning all rows (365 rows in `data_src` and 93,845 rows in `datsrcln`).
  Hash Joins introduced additional overhead for creating hash tables on large intermediate results making high execution time of 11.188 ms.

- **After: Index Only Scans and Nested Loop Joins**
  The partial index on `vol_city` allowed the query to directly locate rows matching `'Cincinnati'` without scanning irrelevant data. Index Only Scan on `datsrcln` reduced heap fetches and narrowed down rows early in the execution plan. Covering indexes on `datsrcln` and `nutr_def` enabled fetching required columns (`tagname` and `ndb_no`) directly from indexes, eliminating additional table lookups. Nested Loop Joins replaced Hash Joins, reducing memory and computation overhead for small subsets of filtered data reducing the overall execution to 0.062 ms.

Reduced Execution Time

Execution time improved from 11.188 ms to 0.062 ms due to:

- Fewer rows scanned:
- Rows in `data_src` reduced from 365 to 1 (via partial index).
- Rows in `datsrcln` reduced from 93,845 to 27 (via composite and covering indexes).
- Efficient join operations:
  Index Only Scans reduced table access and intermediate computations.
  Nested Loop Joins were faster than Hash Joins for the small filtered dataset.
- **Index Only Scan benefits:** Eliminated the need for heap accesses, resulting in reduced I/O and overall faster query execution.

Result: Significant Query Optimization

Query: 14
```
Select * from weight where amount < 2;
```

Attribute:
Btree index on `amount` in `weight`
Partial index on `amount` in `weight WHERE amount < 2`

Most expedient index:
USED: `CREATE INDEX idx_amount_btree ON weight (amount);`
USED: `CREATE INDEX idx_partial_amount ON weight (amount) WHERE amount < 2;`

Effect of the index:

	Before	**After**
Planning time	0.061 ms	0.078 ms
Execution time	1.827 ms	1.382 ms
Highest cost	Seq Scan on weight = 288	Bitmap heap Scan on weight = 273
Slowest runtime	Seq Scan on weight = 1.36 ms	Bitmap heap Scan on weight = 0.810 ms
Largest number of rows	Seq Scan on weight = 11,625 rows	Bitmap Index Scan on weight = 11,625 rows Bitmap heap Scan on weight = 11,625 rows
Query Planner	https://explain.dalibo.com/plan/adbgcb902hh83af1	https://explain.dalibo.com/plan/h709g68c36954a98

Plan before:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on weight (cost=0.00..287.61 rows=11624 width=50) (actual time=0.008..1.362 rows=11625 loops=1)
2	Filter: (amount < '2'::double precision)
3	Rows Removed by Filter: 1384
4	Planning Time: 0.061 ms
5	Execution Time: 1.827 ms

Plan after:

```
8  set enable_seqscan = off
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on weight (cost=197.31..467.61 rows=11624 width=50) (actual time=0.215..1.009 rows=11625 loops=1)
2	Recheck Cond: (amount < '2'::double precision)
3	Heap Blocks: exact=125
4	-> Bitmap Index Scan on idx_partial_amount (cost=0.00..194.41 rows=11624 width=0) (actual time=0.198..0.199 rows=11625 loops=1)
5	Planning Time: 0.078 ms
6	Execution Time: 1.382 ms

Justification:

Index and Type Justification

- **B-tree Index on `amount` in `weight`:**
  Optimized for queries involving range conditions (<, >, etc.).

- **Partial Index on `amount` in `weight` `WHERE amount < 2`:**
  Targets only rows matching the condition `amount < 2`, significantly reducing the scanned dataset size.

Techniques Disabled: `set enable_seqscan= off;`

Sequential Scan (Before) vs. Optimized Indexing (After)

- **Before: Sequential Scan**
  The sequential scan processed all rows (13,009 in total), removing 1,384 rows that did not satisfy the condition `amount < 2`. Resulted in higher execution time due to the need to examine every row, even those outside the desired range.

- **After: Bitmap Index Scan**
  The partial index was utilized **after sequential scans were disable**d, significantly reducing heap fetches and focusing only on relevant rows (11,625 rows). A Bitmap Index Scan efficiently identified matching rows while minimizing table access.

Reduced Execution Time
Execution time improved from **1.827 ms to 1.382 ms** due to:

- **Focused filtering:** The partial index allowed targeting only rows satisfying `amount < 2`.
- **Efficient scan strategy:** The Bitmap Index Scan avoided scanning irrelevant data blocks.

Result: Partial Optimization
The query optimization was successful under specific constraints (disabling sequential scans) but may not generalize for all scenarios due to the **low selectivity** of the condition where the query output 11,625 rows out of 13,009 which is **89.36%**. In general, it is not considered a very highly significant optimization since initially the planner saw no use in optimizing it due to the large number of rows returned.

Query: 15
```
Select * from weight where amount > 2;
```

Attribute:
Btree index on `amount` in `weight`

Most expedient index:
```
CREATE INDEX idx_amount_btree ON weight (amount);
```

Effect of the index:

	Before	**After**
Planning time	0.270 ms	0.072 ms
Execution time	1.327 ms	0.261 ms
Highest cost	Seq Scan on weight = 288	Bitmap heap Scan on weight = 140
Slowest runtime	Seq Scan on weight = 1.26 ms	Bitmap heap Scan on weight = 0.172 ms
Largest number of rows	Seq Scan on weight = 1,215 rows	Bitmap Index Scan on weight = 1,215 rows Bitmap heap Scan on weight = 1,215 rows
Query Planner	https://explain.dalibo.com/plan/fgbffg8h28169b52	https://explain.dalibo.com/plan/83a559df0ehg917f

Plan before:

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Seq Scan on weight (cost=0.00..287.61 rows=1215 width=50) (actual time=0.045..1.263 rows=1215 loops=1)
2	Filter: (amount > '2'::double precision)
3	Rows Removed by Filter: 11794
4	Planning Time: 0.270 ms
5	Execution Time: 1.327 ms

Plan after:

	QUERY PLAN text
1	Bitmap Heap Scan on weight (cost=25.70..165.89 rows=1215 width=50) (actual time=0.050..0.212 rows=1215 loops=1)
2	Recheck Cond: (amount > '2'::double precision)
3	Heap Blocks: exact=86
4	-> Bitmap Index Scan on idx_amount_btree (cost=0.00..25.40 rows=1215 width=0) (actual time=0.039..0.040 rows=1215 loops=1)
5	Index Cond: (amount > '2'::double precision)
6	Planning Time: 0.072 ms
7	Execution Time: 0.261 ms

Justification:

Index and Type Justification

- **B-tree Index on `amount` in `weight`:**
  Optimized for queries involving range conditions (<, >, etc.).

Sequential Scan (Before) vs. Bitmap Index Scan (After)

- **Before: Sequential Scan**
  The sequential scan processed all rows (13,009 in total), removing 11,794 rows that did not satisfy the condition `amount > 2`. Resulted in higher execution time due to the need to examine every row, even those outside the desired range. This approach was bad because the query has a very **high selectivity retrieving 1,215 rows = 9.34%.**

- **After: Bitmap Index Scan**
  With the addition of the B-tree index, the query used a Bitmap Index Scan directly without needing configuration adjustments. The scan efficiently identified and fetched only the matching rows (1,215), minimizing heap accesses. Bitmap index is useful since it retrieves all relevant rows using bits.

Reduced Execution Time

Execution time improved from **1.327 ms to 0.261 ms** due to:

- **High selectivity:**
  The condition `amount > 2` returned a small subset of rows (1,215 out of 13,009).
- **Efficient scan strategy:**
  The Bitmap Index Scan precisely targeted the relevant rows, significantly reducing the number of blocks accessed.

Result: Fully Optimized Query
The query optimization was highly effective, as the index was utilized without requiring any adjustments. The high selectivity of the condition ensured that the index significantly reduced the execution time and resource usage. In comparison to query 14, we can see how we didn't need to turn anything off for the planner to use it since the query is very highly selective and only selects a small subset. When the query is low-selective like query 14, we forced the planner to use it and even after using it, the optimization was not as huge as in query 15.

Query: 16

```
Select msre_desc, avg (amount)
From weight where amount > 2
Group by amount, msre_desc;
```

Attribute:

Partial composite Index on `(amount, msre_desc)` in `weight.`
Composite Index on `(amount, msre_desc)` in `weight.`
Covering composite Index on `(amount, msre_desc)` in `weight.`
Partial on `msre_desc` in `weight.`
Partial `amount` in `weight.`

Most expedient index:

```
USED: CREATE INDEX idx_amount_msre_desc_partial ON weight (amount, msre_desc)
 WHERE amount > 2;
USED: CREATE INDEX idx_amount_msre_desc ON weight (amount, msre_desc);
USED: CREATE INDEX idx_covering_amount_msre_desc ON weight (amount, msre_desc)
INCLUDE (amount);
USED: CREATE INDEX idx_partial_msre_desc_2 ON weight (msre_desc) WHERE amount
> 2;
USED: CREATE INDEX idx_partial_amount_gt_2 ON weight (amount)
WHERE amount > 2;
```

Effect of the index:

	Before	**After**
Planning time	0.475 ms	0.226 ms
Execution time	1.797 ms	0.222 ms
Highest cost	Seq Scan on weight = 288	Index Only Scan on weight = 46.5
Slowest runtime	Seq Scan on weight = 1.37 ms	GroupAggregate = 0.109 ms
Largest number of rows	Seq Scan on weight = 1,215 rows	Index Only Scan on weight = 1,215 rows
Query Planner	https://explain.dalibo.com/plan/4085ghh3582a3a74	https://explain.dalibo.com/plan/2a9e5g23h6695bg1

Plan before:

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	HashAggregate (cost=296.73..307.56 rows=867 width=29) (actual time=1.634..1.654 rows=140 loops=1)
2	Group Key: amount, msre_desc
3	Batches: 1 Memory Usage: 73kB
4	-> Seq Scan on weight (cost=0.00..287.61 rows=1215 width=21) (actual time=0.047..1.366 rows=1215 loops=1)
5	Filter: (amount > '2'::double precision)
6	Rows Removed by Filter: 11794
7	Planning Time: 0.475 ms
8	Execution Time: 1.797 ms

Plan after: (Partial composite)

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=0.28..66.45 rows=867 width=29) (actual time=0.017..0.194 rows=140 loops=1)
2	Group Key: amount, msre_desc
3	-> Index Only Scan using idx_amount_msre_desc_partial on weight (cost=0.28..46.50 rows=1215 width=21) (actual time=0.011..0.085 rows=1215 loops=1)
4	Heap Fetches: 0
5	Planning Time: 0.226 ms
6	Execution Time: 0.222 ms

Justification:

Attribute for Index and Most Expedient Index Type Justification

• **Partial B-tree Index on (`amount, msre_desc`) in `weight WHERE amount > 2`:**
Optimized for queries involving range conditions (>) combined with grouping and aggregation. The partial index reduces the scanned dataset to only rows satisfying `amount > 2`, improving filtering and grouping efficiency.

Sequential Scan (Before) vs. Index Only Scan (After)

• **Before: Sequential Scan**
The sequential scan processed all rows (13,009), removing 11,794 rows that did not satisfy the condition `amount > 2`. Resulted in higher execution time due to the need to examine every row. This approach was inefficient because the query condition had **high selectivity**, retrieving **1,215** rows, approximately **9.34%** of the total rows.

• **After: Index Only Scan**
With the addition of the partial B-tree index, the query utilized an Index Only Scan directly. This scan efficiently targeted the relevant rows (1,215), avoiding unnecessary heap fetches. The index also supported the `GROUP BY` operation by **pre-sorting** the indexed columns, further improving execution efficiency. Even when comparing the cost, we can see moving from seqscan (288) to index only scan (46.5) really made the execution faster and more efficient.

Reduced Execution Time

Execution time improved from **1.797** ms to **0.222** ms due to:

- **High selectivity:** The condition `amount > 2` returned only **1,215** rows, allowing the index to focus on a small subset of the dataset.
- **Efficient grouping strategy:** The pre-sorted indexed columns (`amount, msre_desc`) supported faster aggregation with reduced memory usage.
- **Index Only Scan benefits:** Eliminated the need for heap accesses, resulting in reduced I/O and overall faster query execution.

Result: Fully Optimized Query

The query optimization was highly effective, as the partial index was utilized directly without requiring configuration adjustments. The high selectivity and proper indexing strategy ensured significant improvements in execution time and resource utilization.

Alaa Ashraf

Queries [17, 24]

Query: 17

```
select fd.long_desc, nd.num_studies
from food_des fd inner join nut_data nd on fd.ndb_no = nd.ndb_no
group by fd.long_desc,nd.num_studies
having num_studies = (select max(num_studies) from nut_data);
```

Attribute:

Index on `(num_studies)` in `nut_data`

Most expedient index:

```
CREATE INDEX idx_nut_data_num_studies ON nut_data(num_studies);
```

Effect of the index:

	Before	**After**
Planning time	0.566 ms	0.211 ms
Execution time	46.458 ms	3.034 ms
Highest cost	Group by by fd.long_desc, nd.num_studies = 5740	Bitmap Heap Scan on nut_data as nd = 2570
Slowest runtime	Gather Merge = 37.3 ms	Seq Scan on food_des = 1.4 ms Hash on food_des = 1.4 ms
Largest number of rows	Parallel Seq Scan on nut_data = 253824 rows	Seq Scan on food_des = 7146
Query Planner	https://explain.dalibo.com/plan/4883cha921cdc264	https://explain.dalibo.com/plan/g2dc03e8c9639b16

Plan before:

	QUERY PLAN text
1	Group (cost=11848.01..11968.63 rows=1641 width=58) (actual time=44.228..46.345 rows=1 loops=1)
2	Group Key: fd.long_desc, nd.num_studies
3	InitPlan 1 (returns $1)
4	-> Finalize Aggregate (cost=5737.47..5737.48 rows=1 width=4) (actual time=33.509..33.556 rows=1 loops=1)
5	-> Gather (cost=5737.36..5737.47 rows=1 width=4) (actual time=33.297..33.544 rows=2 loops=1)
6	Workers Planned: 1
7	Workers Launched: 1
8	-> Partial Aggregate (cost=4737.36..4737.37 rows=1 width=4) (actual time=27.687..27.689 rows=1 loops=2)
9	-> Parallel Seq Scan on nut_data (cost=0.00..4364.09 rows=149309 width=4) (actual time=0.010..11.223 rows=1...
10	-> Gather Merge (cost=6110.53..6226.33 rows=965 width=58) (actual time=44.226..46.294 rows=1 loops=1)
11	Workers Planned: 1
12	Params Evaluated: $1
13	Workers Launched: 1
14	-> Group (cost=5110.52..5117.75 rows=965 width=58) (actual time=9.004..9.006 rows=0 loops=2)
15	Group Key: fd.long_desc, nd.num_studies
16	-> Sort (cost=5110.52..5112.93 rows=965 width=58) (actual time=9.002..9.004 rows=0 loops=2)
17	Sort Key: fd.long_desc
18	Sort Method: quicksort Memory: 25kB
19	Worker 0: Sort Method: quicksort Memory: 25kB
20	-> Hash Join (cost=322.79..5062.68 rows=965 width=58) (actual time=6.809..8.963 rows=0 loops=2)
21	Hash Cond: (nd.ndb_no = fd.ndb_no)
22	-> Parallel Seq Scan on nut_data nd (cost=0.00..4737.36 rows=965 width=10) (actual time=5.735..7.887 rows=...
23	Filter: (num_studies = $1)
24	Rows Removed by Filter: 126912
25	-> Hash (cost=233.46..233.46 rows=7146 width=60) (actual time=2.040..2.040 rows=7146 loops=1)
26	Buckets: 8192 Batches: 1 Memory Usage: 708kB
27	-> Seq Scan on food_des fd (cost=0.00..233.46 rows=7146 width=60) (actual time=0.022..0.847 rows=7146...
28	Planning Time: 0.566 ms
29	Execution Time: 46.458 ms

Plan after:

	QUERY PLAN text
1	HashAggregate (cost=2930.40..2946.81 rows=1641 width=58) (actual time=2.900..2.935 rows=1 loops=1)
2	Group Key: fd.long_desc, nd.num_studies
3	Batches: 1 Memory Usage: 73kB
4	InitPlan 2 (returns $1)
5	-> Result (cost=0.44..0.45 rows=1 width=4) (actual time=0.008..0.012 rows=1 loops=1)
6	InitPlan 1 (returns $0)
7	-> Limit (cost=0.42..0.44 rows=1 width=4) (actual time=0.007..0.010 rows=1 loops=1)
8	-> Index Only Scan Backward using idx_nut_data_num_studies on nut_data (cost=0.42..552.00 rows=26262 width=4) (...
9	Index Cond: (num_studies IS NOT NULL)
10	Heap Fetches: 0
11	-> Hash Join (cost=343.92..2921.74 rows=1641 width=58) (actual time=2.846..2.865 rows=1 loops=1)
12	Hash Cond: (nd.ndb_no = fd.ndb_no)
13	-> Bitmap Heap Scan on nut_data nd (cost=21.14..2594.65 rows=1641 width=10) (actual time=0.016..0.030 rows=1 loops=1)
14	Recheck Cond: (num_studies = $1)
15	Heap Blocks: exact=1
16	-> Bitmap Index Scan on idx_nut_data_num_studies (cost=0.00..20.73 rows=1641 width=0) (actual time=0.013..0.013 ro...
17	Index Cond: (num_studies = $1)
18	-> Hash (cost=233.46..233.46 rows=7146 width=60) (actual time=2.798..2.799 rows=7146 loops=1)
19	Buckets: 8192 Batches: 1 Memory Usage: 708kB
20	-> Seq Scan on food_des fd (cost=0.00..233.46 rows=7146 width=60) (actual time=0.004..1.397 rows=7146 loops=1)
21	Planning Time: 0.211 ms
22	Execution Time: 3.034 ms

Justification

Justification for attribute:
The query involves a join between two large tables, `food_des` and `nut_data`, with filtering on `num_studies`. The filtering attribute (`num_studies`) is critical to reduce the dataset before joining. By creating an index on `num_studies`, the query can efficiently locate the row with the maximum value. This optimization reduces the cost of sequential scans over the large `nut_data` table.

Justification for index:
A B+Tree Index on `num_studies` was used since it efficiently supports aggregates, range queries and exact matches. This index is suitable for operations such as finding the maximum value, which is crucial for the subquery in the `HAVING` clause.

Justification for observed behavior:

- **Sequential Scan (Before) vs. Index Scan (After) on `nut_data`:**
    - ***Before:*** A sequential scan was used to evaluate all **253,824** rows in `nut_data` for filtering `num_studies`. This was computationally expensive and increased the query's execution time.
    - ***After:*** The index scan allowed direct access to the row with the maximum `num_studies` value, eliminating the need to scan all rows. So only 1 row was fetched from nut_data with an Index Only scan backward. The parallel Seq Scan used for the Hash join with food_des also got eliminated and replaced with a very efficient Bitmap Index Scan with a fraction of the cost.
- **Group Operation (Before) vs. Hash Aggregate (After):**
    - ***Before:*** The `Group` operation with sorting was performed on a large dataset. This involved sorting intermediate results, consuming more memory and time.
    - ***After:*** The `HashAggregate` operation efficiently grouped and aggregated the data in memory without extensive sorting, leveraging the reduced dataset from the indexed filter.
- **Execution Time Improvements:**
    - Execution time improved significantly, from **46.458 ms** to **3.034 ms**, due to:
        1. Efficient filtering with the index on `num_studies` (reducing rows from **253,824** to **1**).
        2. Elimination of expensive sequential scans.
        3. Faster aggregation enabled by the reduced intermediate dataset.

Result:
Highly significant query optimization.

Query: 18

```
select * from fd_group fd inner join food_des fod on fd.fdgrp_cd =
fod.fdgrp_cd;
```

Attempted Attribute:

Index on `(fdgrp_cd)` in `food_des`

Attempted index:

```
CREATE INDEX idx_fooddes_fdgrp_cd ON food_des(fdgrp_cd);
```

Effect of the index:

	Before	**After**
Planning time	0.113 ms	0.192 ms
Execution time	1.954 ms	2.021 ms
Highest cost	Seq Scan on food_des = 233	Index Scan on food_des = 420.45
Slowest runtime	Hash Join on fod.fdgrp_cd = fd.fdgrp_cd = 1.41 ms	Merge Join on fod.fdgrp_cd = fd.fdgrp_cd = 1.12 ms
Largest number of rows	Seq Scan on food_des = 7146 rows	Index Scan on food_des = 7146 rows
Query Planner	https://explain.dalibo.com/plan/7528bhc48bfh1e83	https://explain.dalibo.com/plan/dc92eca8125a2ce1

Plan before:

	QUERY PLAN text
1	Hash Join (cost=1.54..257.07 rows=7146 width=175) (actual time=0.018..1.797 rows=7146 loops=1)
2	Hash Cond: (fod.fdgrp_cd = fd.fdgrp_cd)
3	-> Seq Scan on food_des fod (cost=0.00..233.46 rows=7146 width=151) (actual time=0.004..0.375 rows=7146 loops=1)
4	-> Hash (cost=1.24..1.24 rows=24 width=24) (actual time=0.007..0.008 rows=24 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 10kB
6	-> Seq Scan on fd_group fd (cost=0.00..1.24 rows=24 width=24) (actual time=0.001..0.002 rows=24 loops=1)
7	Planning Time: 0.113 ms
8	Execution Time: 1.954 ms

Plan after:

	QUERY PLAN text
1	Merge Join (cost=0.42..522.34 rows=7146 width=175) (actual time=0.015..1.862 rows=7146 loops=1)
2	Merge Cond: (fd.fdgrp_cd = fod.fdgrp_cd)
3	-> Index Scan using fd_group_pkey on fd_group fd (cost=0.14..12.50 rows=24 width=24) (actual time=0.003..0.007 ro...
4	-> Index Scan using idx_fooddes_fdgrp_cd on food_des fod (cost=0.28..420.45 rows=7146 width=151) (actual time=0....
5	Planning Time: 0.192 ms
6	Execution Time: 2.021 ms

Justification

Justification for attributes: The query joins `fd_group` and `food_des` tables on the `fdgrp_cd` attribute. This attribute is essential as it acts as the primary key in `fd_group` and a foreign key in `food_des`, establishing the relationship between the two tables. Efficient filtering and joining depend on indexing this attribute.

Justification for index: An index `(idx_fooddes_fdgrp_cd)` was created on `food_des(fdgrp_cd)` because `fdgrp_cd` is frequently used in join conditions. Using a B+Tree index usually enables faster lookups.

Techniques Disabled: `set enable_hashjoin= off; set enable_seqscan= off;`

Justification for observed behavior:

- **Sequential Scan (Before) vs. Index Scan (After):** First a Sequential Scan on food_des for all **7146** rows was used to fetch at the records for the join. It may seem that Index Scan might improve this, but it has not done so because all the rows are needed anyway for the join and there is no filtering in the query. Therefore, when fetching all the records it’s definitely faster to do Sequential scan than to go to each record's position in the index and waste compute time then again to the database store as observed by the increase in operation cost between the Seq Scan and Index scan from **233** to **420**.
- **Hash Join (Before) vs Merge Join (After)**: This method uses hashing for the join operation, it relies on sequential scans for both tables. After indexing and disabling Seq Scan and Hash Join, the query used a Merge Join.
- **Execution Time**: The execution time slightly increased from **1.954** ms to **2.021** ms

Result: No significant optimization observed.

Query: 19:

```
select * from fd_group fd inner join food_des fod on fd.fdgrp_cd =
fod.fdgrp_cd inner join nut_data nd on nd.ndb_no = fod.ndb_no;
```

Attribute:

Index on `(ndb_no)` in `nut_data`
Index on `(fdgrp_cd)` in `food_des`

Most expedient index:

USED: `CREATE INDEX idx_nut_data_ndb_no ON nut_data(ndb_no);`
USED: `CREATE INDEX idx_fooddes_fdgrp_cd ON food_des(fdgrp_cd);`

Effect of the index:

	Before	**After**
Planning time	0.227 ms	0.238 ms
Execution time	141.721 ms	89.976 ms
Highest cost	Seq Scan on nut_data = 5410	Index Scan on nut_data as nd using idx_nut_data_ndb_no = 7601.67
Slowest runtime	Hash Join on nd.ndb_no = fod.ndb_no = 76 ms	Merge Join on nd.ndb_no = fod.ndb_no = 47.9 ms
Largest number of rows	Seq Scan on nut_data = 253825 rows	Index Scan on nut_data = 253825 rows
Query Planner	https://explain.dalibo.com/plan/2da7dgca407gdfdc	https://explain.dalibo.com/plan/65450d2ggeb65a54

Plan before:

	QUERY PLAN text
1	Hash Join (cost=324.33..7184.21 rows=253825 width=267) (actual time=1.134..135.999 rows=253825 loops=1)
2	Hash Cond: (fod.fdgrp_cd = fd.fdgrp_cd)
3	-> Hash Join (cost=322.79..6398.73 rows=253825 width=243) (actual time=1.120..92.199 rows=253825 loops=1)
4	Hash Cond: (nd.ndb_no = fod.ndb_no)
5	-> Seq Scan on nut_data nd (cost=0.00..5409.25 rows=253825 width=92) (actual time=0.002..15.075 rows=253825...
6	-> Hash (cost=233.46..233.46 rows=7146 width=151) (actual time=1.105..1.107 rows=7146 loops=1)
7	Buckets: 8192 Batches: 1 Memory Usage: 1364kB
8	-> Seq Scan on food_des fod (cost=0.00..233.46 rows=7146 width=151) (actual time=0.002..0.374 rows=7146 l...
9	-> Hash (cost=1.24..1.24 rows=24 width=24) (actual time=0.009..0.010 rows=24 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 10kB
11	-> Seq Scan on fd_group fd (cost=0.00..1.24 rows=24 width=24) (actual time=0.003..0.004 rows=24 loops=1)
12	Planning Time: 0.227 ms
13	Execution Time: 141.721 ms

Plan after:

	QUERY PLAN text
1	Merge Join (cost=980.08..11790.00 rows=253825 width=267) (actual time=10.011..84.566 rows=253825 loops=1)
2	Merge Cond: (fod.ndb_no = nd.ndb_no)
3	-> Sort (cost=979.78..997.65 rows=7146 width=175) (actual time=9.978..10.359 rows=7146 loops=1)
4	Sort Key: fod.ndb_no
5	Sort Method: quicksort Memory: 2230kB
6	-> Merge Join (cost=0.42..522.34 rows=7146 width=175) (actual time=0.012..2.141 rows=7146 loops=1)
7	Merge Cond: (fd.fdgrp_cd = fod.fdgrp_cd)
8	-> Index Scan using fd_group_pkey on fd_group fd (cost=0.14..12.50 rows=24 width=24) (actual time=0.002..0....
9	-> Index Scan using idx_fooddes_fdgrp_cd on food_des fod (cost=0.28..420.45 rows=7146 width=151) (actual ti...
10	-> Index Scan using idx_nut_data_ndb_no on nut_data nd (cost=0.29..7601.67 rows=253825 width=92) (actual time=0....
11	Planning Time: 0.238 ms
12	Execution Time: 89.976 ms

Justification

Justification for attributes: This query involves joins between `fd_group`, `food_des`, and `nut_data` tables. The critical attributes are:

- `fdgrp_cd`: For joining `fd_group` and `food_des`.
- `ndb_no`: For joining `food_des` and `nut_data`.

Efficient access to these attributes is crucial to reduce scanning overhead and improve joins.

Justification for indexes:

- Using a B+Tree index usually enables faster lookups, which is what is needed here in our query.

Techniques Disabled: `set enable_hashjoin= off; set enable_seqscan= off;`

Justification for observed behavior:

- **Hash Join & Seq Scan (Before)**: Both joins used hash-based operations, requiring sequential scans on `food_des` and `nut_data`, the Planner here also decided to do larger joins first instead of the smallest first which greatly increased the execution time.
- **Merge Join & Index Scan (After)**: By using the indexes, the query transitioned to Merge Joins, leveraging sorted data. The `nut_data` index reduced the sequential scan of **253,825** rows to an index scan and the planner decided to then do the smallest joins first, significantly lowering the execution time from **141.721** ms to **89.976** ms.

Result: Significant query optimization.

Query: 20

```
(select ndb_no from weight where amount > 50) INTERSECT (select ndb_no from
food_des fod inner join fd_group fd on fd.fdgrp_cd = fod.fdgrp_cd where
fd.fddrp_desc = 'Snacks');
```

Attribute:

Index on `(fdgrp_cd)` in `food_des`
Index on `(ndb_no)` in `weight`

Most expedient index:

USED: `CREATE INDEX idx_weight_ndb_no ON weight(ndb_no) WHERE amount > 50;`
USED: `CREATE INDEX idx_fooddes_fdgrp_cd ON food_des(fdgrp_cd);`

Effect of the index:

	Before	After
Planning time	0.251 ms	0.165 ms
Execution time	2.225 ms	0.083 ms
Highest cost	Seq Scan on weight = 288	Index Scan on food_des using idx_fooddes_fdgrp_cd = 162
Slowest runtime	Seq Scan on weight = 0.814 ms	Index Scan on food_des using idx_fooddes_fdgrp_cd = 0.03 ms
Largest number of rows	Seq Scan on food_des = 7146 rows	Index Scan on food_des using idx_fooddes_fdgrp_cd = 118 rows
Query Planner	https://explain.dalibo.com/plan/17e156d6b88agea3	https://explain.dalibo.com/plan/bd3cc8533d94c349

Plan before:

	QUERY PLAN text
1	HashSetOp Intersect (cost=0.00..549.69 rows=1 width=28) (actual time=2.157..2.164 rows=0 loops=1)
2	-> Append (cost=0.00..548.94 rows=299 width=28) (actual time=1.814..2.156 rows=118 loops=1)
3	-> Subquery Scan on "*SELECT* 1" (cost=0.00..287.62 rows=1 width=10) (actual time=0.815..0.816 rows=0 loops=1)
4	-> Seq Scan on weight (cost=0.00..287.61 rows=1 width=6) (actual time=0.814..0.814 rows=0 loops=1)
5	Filter: (amount > '50'::double precision)
6	Rows Removed by Filter: 13009
7	-> Subquery Scan on "*SELECT* 2" (cost=1.31..259.83 rows=298 width=10) (actual time=0.996..1.330 rows=118 loops=1)
8	-> Hash Join (cost=1.31..256.85 rows=298 width=6) (actual time=0.995..1.321 rows=118 loops=1)
9	Hash Cond: (fod.fdgrp_cd = fd.fdgrp_cd)
10	-> Seq Scan on food_des fod (cost=0.00..233.46 rows=7146 width=11) (actual time=0.018..0.554 rows=7146 loops=1)
11	-> Hash (cost=1.30..1.30 rows=1 width=5) (actual time=0.014..0.016 rows=1 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Seq Scan on fd_group fd (cost=0.00..1.30 rows=1 width=5) (actual time=0.009..0.010 rows=1 loops=1)
14	Filter: (fddrp_desc = 'Snacks'::text)
15	Rows Removed by Filter: 23
16	Planning Time: 0.251 ms
17	Execution Time: 2.225 ms

Plan after:

	QUERY PLAN text
1	HashSetOp Intersect (cost=0.12..175.50 rows=1 width=28) (actual time=0.062..0.063 rows=0 loops=1)
2	-> Append (cost=0.12..174.75 rows=299 width=28) (actual time=0.015..0.057 rows=118 loops=1)
3	-> Subquery Scan on "*SELECT* 1" (cost=0.12..4.15 rows=1 width=10) (actual time=0.002..0.002 rows=0 loops=1)
4	-> Index Only Scan using idx_weight_ndb_no on weight (cost=0.12..4.14 rows=1 width=6) (actual time=0.001..0.001 rows=0 loops=1)
5	Heap Fetches: 0
6	-> Subquery Scan on "*SELECT* 2" (cost=0.28..169.11 rows=298 width=10) (actual time=0.013..0.049 rows=118 loops=1)
7	-> Nested Loop (cost=0.28..166.13 rows=298 width=6) (actual time=0.013..0.043 rows=118 loops=1)
8	-> Seq Scan on fd_group fd (cost=0.00..1.30 rows=1 width=5) (actual time=0.005..0.005 rows=1 loops=1)
9	Filter: (fddrp_desc = 'Snacks'::text)
10	Rows Removed by Filter: 23
11	-> Index Scan using idx_fooddes_fdgrp_cd on food_des fod (cost=0.28..161.85 rows=298 width=11) (actual time=0.006..0.030 rows=1...
12	Index Cond: (fdgrp_cd = fd.fdgrp_cd)
13	Planning Time: 0.165 ms
14	Execution Time: 0.083 ms

Justification

Justification for attributes: The query applies an `INTERSECT` operation between:

1. `weight.amount > 50`: Filters rows in the `weight` table.
2. `fd_group.fddrp_desc = 'Snacks'`: Filters rows in `fd_group`.

Efficient access to these filtering attributes ensures better query performance. The filtering attributes (`amount` and `fdgrp_cd`) are critical to reduce the dataset size before performing expensive operations like joins and intersections.

Justification for index:
A **B+Tree Index** was created on the `amount` column in the `weight` table (`idx_weight_ndb_no`) with a condition `WHERE amount > 50`. This ensures that only relevant rows are scanned. Similarly, an index on the `fdgrp_cd` column in the `food_des` table (`idx_fooddes_fdgrp_cd`) enables faster lookups when joining with the `fd_group` table. There are 118 resulting rows here so B+ Tree index would be good for this exact value lookup.

Justification for observed behavior:

Sequential Scan (Before) vs. Index Scan (After):

- **Before:** The `weight` table underwent a sequential scan, evaluating all **13,009** rows to filter out rows with an `amount > 50`. This approach was computationally expensive, resulting in higher execution times.
- **After:** The index on `amount` allowed for an **Index Only Scan**, directly accessing the relevant rows without scanning the entire table because the partial index already satisfies `amount > 50`. As a result, no rows were fetched unnecessarily, making the operation faster. The other Seq scan was also prevented on `food_des` due to the index on `fdgrp_cd.`

Hash Join (Before) vs. Nested Loop with Index Scan (After):

- **Before:** A **Hash Join** was used between the `fd_group` and `food_des` tables, requiring a sequential scan of the `food_des` table (**7,146** rows) to build the hash table. This operation incurred high memory usage and longer execution time.
- **After:** A **Nested Loop Join** leveraged the index on `fdgrp_cd` (`idx_fooddes_fdgrp_cd`), which allowed the query to fetch matching rows directly. This significantly reduced the number of rows scanned and improved performance.

Execution Time Improvements:
Execution time improved dramatically, from **2.225 ms to 0.083 ms**, due to:

- Efficient filtering through indexes, reducing scanned rows. (**13009 -> 0**)
- Elimination of costly sequential scans and hash joins.
- Optimized data access through index scans, resulting in minimal intermediate data processing.

Result:
The query is highly optimized.

Query: 21

```
select max(ndb_no) from (select * from fd_group fd inner join food_des fod on
fd.fdgrp_cd = fod.fdgrp_cd) as t
```

Attribute:

Index on `(fdgrp_cd, ndb_no)` in `food_des`

Most expedient index:

```
CREATE INDEX idx_food_des_comp ON food_des(fdgrp_cd, ndb_no);
```

Effect of the index:

	Before	After
Planning time	0.176 ms	0.166 ms
Execution time	3.606 ms	1.371 ms
Highest cost	Seq Scan on food_des = 233	Nested Loop (merge fd & fod) = 301
Slowest runtime	Hash Join on fod.fdgrp_cd = fd.fdgrp_cd = 1.58 ms	Index Only Scan on food_des using idx_food_des_comp = 0.576 ms
Largest number of rows	Seq Scan on food_des = 7146 rows	Index Only Scan on food_des using idx_food_des_comp = 7152 rows
Query Planner	https://explain.dalibo.com/plan/69f7fb982gg2222c	https://explain.dalibo.com/plan/793b389abee82540

Plan before:

	QUERY PLAN text
1	Aggregate (cost=274.94..274.95 rows=1 width=32) (actual time=3.552..3.558 rows=1 loops=1)
2	-> Hash Join (cost=1.54..257.07 rows=7146 width=6) (actual time=0.033..2.268 rows=7146 loops=1)
3	Hash Cond: (fod.fdgrp_cd = fd.fdgrp_cd)
4	-> Seq Scan on food_des fod (cost=0.00..233.46 rows=7146 width=11) (actual time=0.008..0.668 rows=7146 loops...
5	-> Hash (cost=1.24..1.24 rows=24 width=5) (actual time=0.013..0.015 rows=24 loops=1)
6	Buckets: 1024 Batches: 1 Memory Usage: 9kB
7	-> Seq Scan on fd_group fd (cost=0.00..1.24 rows=24 width=5) (actual time=0.005..0.008 rows=24 loops=1)
8	Planning Time: 0.176 ms
9	Execution Time: 3.606 ms

Plan after:

	QUERY PLAN text
1	Aggregate (cost=341.82..341.83 rows=1 width=32) (actual time=1.345..1.346 rows=1 loops=1)
2	-> Nested Loop (cost=0.42..323.96 rows=7146 width=6) (actual time=0.019..0.907 rows=7146 loops=1)
3	-> Index Only Scan using fd_group_pkey on fd_group fd (cost=0.14..12.50 rows=24 width=5) (actual time=0.003..0....
4	Heap Fetches: 24
5	-> Index Only Scan using idx_food_des_comp on food_des fod (cost=0.28..10.00 rows=298 width=11) (actual time...
6	Index Cond: (fdgrp_cd = fd.fdgrp_cd)
7	Heap Fetches: 0
8	Planning Time: 0.166 ms
9	Execution Time: 1.371 ms

Justification

Justification for Attribute

The query involves joining the `food_des` and `fd_group` tables based on the `fdgrp_cd` attribute, with an aggregation operation `max(ndb_no)` performed on the joined dataset. The `fdgrp_cd` column in both tables is critical for this join operation as it determines the matching rows between the tables.

Justification for Index

An **Index on `fdgrp_cd` in `food_des`**, combined with the secondary column `ndb_no`, was created using a B+Tree structure (`idx_food_des_comp`). This type of index is well-suited for supporting equality joins `(fdgrp_cd = fd.fdgrp_cd)` and efficiently narrowing down rows during scans. This multi-column index further improves performance when `ndb_no` is accessed as part of additional operations, reducing unnecessary access.

Justification for Observed Behavior

Sequential Scan (Before) vs. Index Scan (After) on `food_des`:

- **Before:**
  - A **Sequential Scan** on the `food_des` table processed all **7,146** rows, leading to increased computational cost and longer execution times.
  - Rows were evaluated linearly, and no shortcuts existed for directly locating relevant rows like `ndb_no` based on `fdgrp_cd`.
- **After:**
  - The **Index Only Scan** on `food_des` used the composite index (`idx_food_des_comp`).

    - Matching rows were located using the indexed `fdgrp_cd`, with no heap fetches required, reducing time and memory usage since the final result needed is the `max(ndb_no)` which was obtained directly from the index after the index only matching with a nested loop.

Hash Join (Before) vs. Nested Loop with Index Scan (After):

- **Before:**
    - A **Hash Join** required creating an in-memory hash table for the `fd_group` table, with a sequential scan of `food_des` feeding rows into the join process.
    - This approach involved a full table scan and fetch of `fd_group` (**24** rows) and `food_des` (7,146 rows), consuming more resources.
- **After:**
    - A **Nested Loop Join** was implemented, leveraging indexes on both tables.
    - The **Primary Key Index** (`fd_group_pkey`) in `fd_group` enabled an **Index Only Scan**, quickly fetching matching rows.
    - The index on `fdgrp_cd` in `food_des` further optimized the join by enabling targeted scans for each match, reducing intermediate dataset size and eliminating unnecessary rows early, especially since only **1** row is required in the end.

Execution Time Improvements:
Execution time improved from **3.606 ms to 1.371 ms**, driven by:

- Replacement of sequential scans with efficient **Index Only Scans**.
- Reduction in intermediate data processed during joins.
- Faster joins through a **Nested Loop** optimized by indexes.

Result: Significant query optimization.

Query: 22

```
select ndb_no, sum(fat_factor), sum(pro_factor) from (select * from fd_group fd inner join food_des fod on fd.fdgrp_cd = fod.fdgrp_cd) as t group by fat_factor, pro_factor, ndb_no;
```

Attribute:

Attempted

Index on `(fat_factor)` in `food_des`
Index on `(pro_factor)` in `food_des`
Index on `(ndb_no)` in `food_des`
Composite Indexes

Attempted indexes: **(all not used)**

```
CREATE INDEX idx_food_des_fat_factor ON food_des(fat_factor);
CREATE INDEX idx_food_des_pro_factor ON food_des(pro_factor);
CREATE INDEX idx_food_des_sums ON food_des(fat_factor, pro_factor);
CREATE INDEX idx_food_des_all ON food_des(ndb_no, fat_factor, pro_factor);
CREATE INDEX idx_food_des_all_order ON food_des(pro_factor, fat_factor, ndb_no);
```

Effect of the index:

	Before	**After**
Planning time	0.197 ms	0.16 ms
Execution time	6.02 ms	6.01 ms
Highest cost	Seq Scan on food_des = 233.46	Index Scan on food_des using food_des_pkey = 360.47
Slowest runtime	Hash Aggregate by fod.ndb_no = 3.31 ms	Hash Aggregate by fod.ndb_no = 2.98 ms
Largest number of rows	Seq Scan on food_des = 7146 rows	Index Scan on food_des = 7146 rows
Query Planner	https://explain.dalibo.com/plan/2318fcg221ha460b	https://explain.dalibo.com/plan/9e2bg2cef261e84f

Plan before:

	QUERY PLAN text
1	HashAggregate (cost=310.67..382.13 rows=7146 width=38) (actual time=3.820..5.527 rows=7146 loops=1)
2	Group Key: fod.ndb_no
3	Batches: 1 Memory Usage: 1425kB
4	-> Hash Join (cost=1.54..257.07 rows=7146 width=22) (actual time=0.024..2.218 rows=7146 loops=1)
5	Hash Cond: (fod.fdgrp_cd = fd.fdgrp_cd)
6	-> Seq Scan on food_des fod (cost=0.00..233.46 rows=7146 width=27) (actual time=0.007..0.490 rows=7146 loops...
7	-> Hash (cost=1.24..1.24 rows=24 width=5) (actual time=0.008..0.010 rows=24 loops=1)
8	Buckets: 1024 Batches: 1 Memory Usage: 9kB
9	-> Seq Scan on fd_group fd (cost=0.00..1.24 rows=24 width=5) (actual time=0.002..0.004 rows=24 loops=1)
10	Planning Time: 0.197 ms
11	Execution Time: 6.016 ms

Plan after:

	QUERY PLAN text
1	HashAggregate (cost=448.94..520.40 rows=7146 width=38) (actual time=4.424..5.733 rows=7146 loops=1)
2	Group Key: fod.ndb_no
3	Batches: 1 Memory Usage: 1425kB
4	-> Hash Join (cost=13.08..395.34 rows=7146 width=22) (actual time=0.028..2.750 rows=7146 loops=1)
5	Hash Cond: (fod.fdgrp_cd = fd.fdgrp_cd)
6	-> Index Scan using food_des_pkey on food_des fod (cost=0.28..360.47 rows=7146 width=27) (actual time=0.010..1.127 rows=7146 loops=1)
7	-> Hash (cost=12.50..12.50 rows=24 width=5) (actual time=0.011..0.014 rows=24 loops=1)
8	Buckets: 1024 Batches: 1 Memory Usage: 9kB
9	-> Index Only Scan using fd_group_pkey on fd_group fd (cost=0.14..12.50 rows=24 width=5) (actual time=0.003..0.006 rows=24 loops=1)
10	Heap Fetches: 24
11	Planning Time: 0.160 ms
12	Execution Time: 6.010 ms

Justification

Justification for Attributes

This query involves a join between `fd_group` and `food_des` tables with aggregation over critical attributes:

- **`fdgrp_cd`**: Used to join `fd_group` and `food_des`. Efficient access to this attribute is key for the join performance.
- **`ndb_no`**: Acts as the group key for aggregation. Optimized access improves grouping and aggregation performance.

- **`fat_factor` and `pro_factor`**: Columns being aggregated. Indexing these attributes can help reduce aggregation overhead.

Justification for Indexes

Using a **B+Tree index** is beneficial for fast lookups and sorted access. Several indexes were tested:

- **`fat_factor` and `pro_factor`**: To support aggregation functions.
- **Composite indexes** (`fat_factor`, `pro_factor`, `ndb_no`): To optimize group-by operations.
- **Default primary key indexes (`pkey`)**: Efficiently used for joins but not aggregation.

Techniques Disabled

- **`set enable_hashjoin = off;`**: To evaluate index-driven join alternatives.
- **`set enable_seqscan = off;`**: To enforce index utilization for data access.

Justification for Observed Behavior

Hash Join & Seq Scan (Before):

- The join utilized hash-based operations, requiring sequential scans on both `fd_group` and `food_des`.
- Aggregation (`HashAggregate`) was performed after the join, increasing memory and processing overhead.
- Execution time: **6.016 ms**.

Index Scan with Default PKey (After):

- The query planner chose index scans for both tables (`food_des` and `fd_group`), leveraging their primary key indexes.
- Sequential scans were eliminated, but the aggregation step (`HashAggregate`) remained unchanged, as the tested indexes did not align with the `GROUP BY` order.
- The inner join subquery is the same as query 18 which wasn't optimized and here same thing occurs since we need to fetch approximately all the rows, so there will not be significant optimization.
- The type of aggregates is `SUM` which will not reduce/filter the number of rows fetched
- Execution time: **6.010 ms**.

Result: No significant query optimization observed.

Query: 23

```
select max(total_fat_factor), max(total_pro_factor) from (select ndb_no,
sum(fat_factor) as total_fat_factor, sum(pro_factor) as total_pro_factor from
(select * from fd_group fd inner join food_des fod on fd.fdgrp_cd =
fod.fdgrp_cd) as t group by fat_factor, pro_factor, ndb_no) as x;
```

Attribute:

Attempted but not used

Index on `(fat_factor)` in `food_des`
Index on `(pro_factor)` in `food_des`
Index on `(ndb_no)` in `food_des`
Composite Indexes

Most expedient index: (not used)

```
CREATE INDEX idx_food_des_fat_factor ON food_des(fat_factor);
CREATE INDEX idx_food_des_pro_factor ON food_des(pro_factor);
CREATE INDEX idx_food_des_sums ON food_des(fat_factor, pro_factor);
CREATE INDEX idx_food_des_all ON food_des(ndb_no, fat_factor, pro_factor);
CREATE INDEX idx_food_des_all_order ON food_des(pro_factor, fat_factor,
ndb_no);
```

Effect of the index:

	Before	**After**
Planning time	0.175 ms	0.17 ms
Execution time	4.385 ms	5.408 ms
Highest cost	Seq Scan on food_des = 233.46 rows	Index Scan on food_des using food_des_pkey = 360.47 rows
Slowest runtime	Hash Aggregate by fod.ndb_no = 2.36 ms	Hash Aggregate by fod.ndb_no = 2.8 ms
Largest number of rows	Seq Scan on food_des = 7146 rows	Index Scan on food_des = 7146 rows
Query Planner	https://explain.dalibo.com/plan/bcd9229dg88314a3	https://explain.dalibo.com/plan/97h7d17845hg710b

Plan before:

	QUERY PLAN text
1	Aggregate (cost=489.32..489.33 rows=1 width=16) (actual time=4.326..4.329 rows=1 loops=1)
2	-> HashAggregate (cost=310.67..382.13 rows=7146 width=38) (actual time=3.100..4.075 rows=7146 loops=1)
3	Group Key: fod.ndb_no
4	Batches: 1 Memory Usage: 1425kB
5	-> Hash Join (cost=1.54..257.07 rows=7146 width=22) (actual time=0.018..1.715 rows=7146 loops=1)
6	Hash Cond: (fod.fdgrp_cd = fd.fdgrp_cd)
7	-> Seq Scan on food_des fod (cost=0.00..233.46 rows=7146 width=27) (actual time=0.005..0.403 rows=7146 lo...
8	-> Hash (cost=1.24..1.24 rows=24 width=5) (actual time=0.007..0.008 rows=24 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 9kB
10	-> Seq Scan on fd_group fd (cost=0.00..1.24 rows=24 width=5) (actual time=0.002..0.004 rows=24 loops=1)
11	Planning Time: 0.175 ms
12	Execution Time: 4.385 ms

Plan after:

	QUERY PLAN text
1	Aggregate (cost=627.59..627.60 rows=1 width=16) (actual time=5.343..5.347 rows=1 loops=1)
2	-> HashAggregate (cost=448.94..520.40 rows=7146 width=38) (actual time=3.909..5.087 rows=7146 loops=1)
3	Group Key: fod.ndb_no
4	Batches: 1 Memory Usage: 1425kB
5	-> Hash Join (cost=13.08..395.34 rows=7146 width=22) (actual time=0.025..2.283 rows=7146 loops=1)
6	Hash Cond: (fod.fdgrp_cd = fd.fdgrp_cd)
7	-> Index Scan using food_des_pkey on food_des fod (cost=0.28..360.47 rows=7146 width=27) (actual time=0.008..0.960 rows=7146 loops=1)
8	-> Hash (cost=12.50..12.50 rows=24 width=5) (actual time=0.010..0.012 rows=24 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 9kB
10	-> Index Only Scan using fd_group_pkey on fd_group fd (cost=0.14..12.50 rows=24 width=5) (actual time=0.002..0.006 rows=24 loops=1)
11	Heap Fetches: 24
12	Planning Time: 0.170 ms
13	Execution Time: 5.408 ms

Justification

Attributes

The query joins the `fd_group` and `food_des` tables on the `fdgrp_cd` attribute. This attribute is crucial as it serves as the primary key in `fd_group` and a foreign key in `food_des`, forming the relationship between the tables. Proper indexing on `fdgrp_cd` supports efficient joins.

Index Justification

Several indexes were created:

- `idx_food_des_fat_factor` and `idx_food_des_pro_factor`: Target aggregation columns.
- `idx_food_des_all (ndb_no, fat_factor, pro_factor)`: To facilitate faster grouped aggregation.
  However, these indexes were not used due to the query's nested aggregation structure, making them irrelevant for the planner.

Techniques Disabled: `SET enable_seqscan = OFF;`

Observed Behavior

Sequential Scan (Before) vs. Index Scan (After) on `food_des`:
The `Seq Scan` fetched all 7146 rows for the join. Replacing it with an `Index Scan` increased the cost due to redundant lookups and data access operations, as all rows are required for the join. After disabling seqscan the planner still didn’t use any of the indexes that were created, however it had to use the default primary key index to change one operation from seqscan to index only scan for small table and index scan for large table since we fetched all rows in both tables anyway. Using an index added overhead in our case, which resulted in no significant optimization. We have also explored turning off hash join; in this case it used merge join and led to no optimization either.

Execution Time

The execution time increased slightly from **4.385 ms** to **5.408 ms**, indicating no performance gain.

Result

No significant optimization observed.

Query: 24

```
select srccd_desc from src_cd;
```

Attribute:

Index on `(srccd_desc)` in `src_cd`

Most expedient index:

```
CREATE INDEX idx_src_cd_srccd_desc ON src_cd(srccd_desc);
```

Effect of the index:

	Before	After
Planning time	0.034 ms	0.157 ms
Execution time	0.021 ms	0.51 ms
Highest cost	Seq Scan on src_cd = 1.1	Index Only Scan on src_cd = 12.1
Slowest runtime	Seq Scan on src_cd = 0.009 rows	Index Only Scan on src_cd = 0.041 rows
Largest number of rows	Seq Scan on src_cd = 10 rows	Index Only Scan on src_cd = 10 rows
Query Planner	https://explain.dalibo.com/plan/052dch873fb349fc	https://explain.dalibo.com/plan/7bfe247g62d9caa7

Plan before:

	QUERY PLAN text
1	Seq Scan on src_cd (cost=0.00..1.10 rows=10 width=44) (actual time=0.008..0.009 rows=10 loops=1)
2	Planning Time: 0.034 ms
3	Execution Time: 0.021 ms

Plan after:

```
set enable_seqscan = off
```

	QUERY PLAN text
1	Index Only Scan using idx_src_cd_srccd_desc on src_cd (cost=0.14..12.29 rows=10 width=44) (actual ...
2	Heap Fetches: 10
3	Planning Time: 0.157 ms
4	Execution Time: 0.051 ms

Justification

Justification for Attribute

The query fetches all rows from the `src_cd` table by selecting the column `srccd_desc`. Since no filtering criteria (e.g., WHERE clause) are applied, the query essentially retrieves all rows in the table.

Justification for Index

An **Index on `srccd_desc`** was created to potentially improve queries filtering or sorting based on this column. However, in this case, the query does not leverage filtering, sorting, or range-based retrieval, which nullifies the advantages of the index. Consequently, the indexed approach introduces unnecessary overhead.

Techniques Disabled: `SET enable_seqscan = OFF;`

Justification for Observed Behavior

Sequential Scan (Before) vs. Index Scan (After):

- **Before:**
  - The **Sequential Scan** simply reads all rows in `src_cd` linearly. With only 10 rows, this operation completed very quickly without additional overhead.
  - Sequential scans are typically faster for select * as they avoid index lookup costs.
- **After:**
  - The **Index Only Scan** involved looking up rows in the index and then performing **Heap Fetches** (10 in total), which added unnecessary steps.
  - The index added an extra layer of complexity, increasing both planning and execution times for a task that sequential scanning could handle more efficiently.

Side note: I did this query in project one and I made a huge table with millions of records to prove that it will never be improved by an index.

Execution Time Impact:
Execution time increased slightly from **0.021 ms to 0.051 ms**, caused by:

- Additional overhead of accessing and traversing the index.
- Heap fetches, which were redundant for such a small dataset.
- Longer planning time (from **0.034 ms** to **0.157 ms**), reflecting the added complexity.

Result: No query optimization