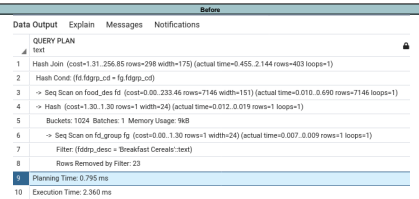
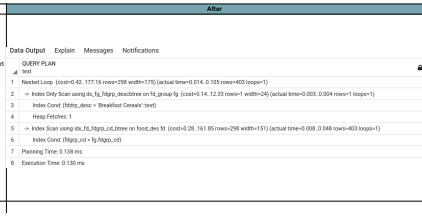
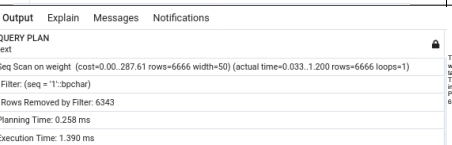
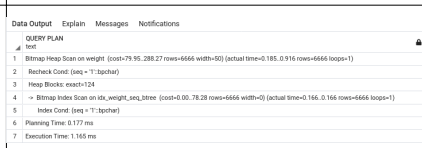
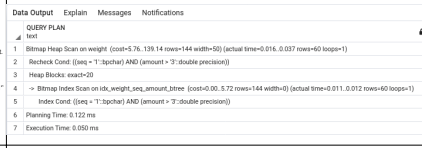
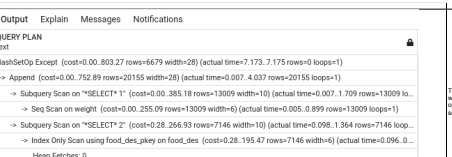
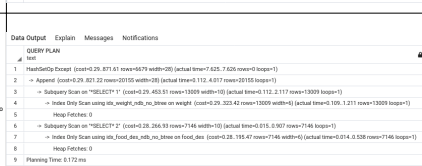
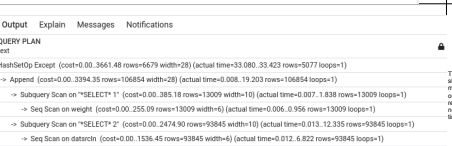
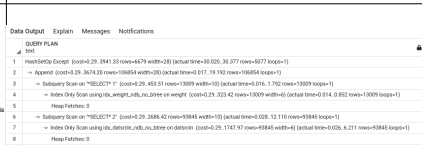
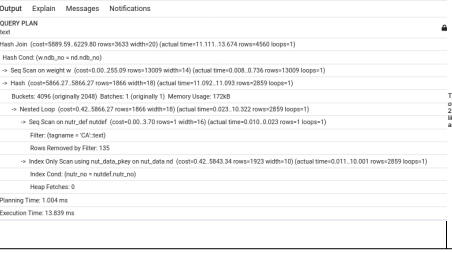
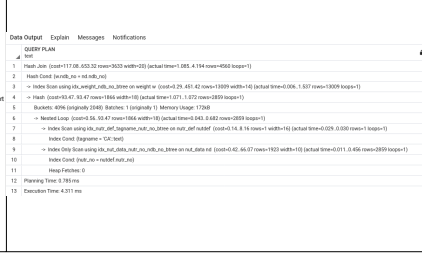
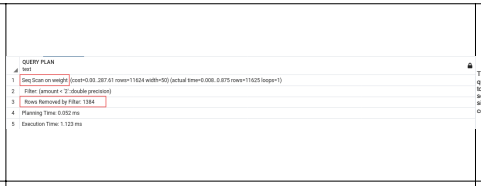
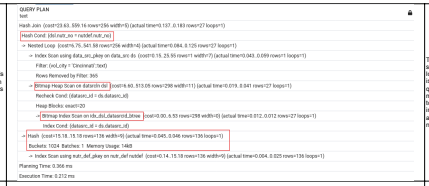
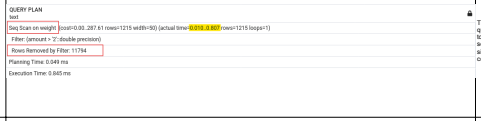
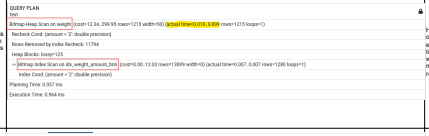
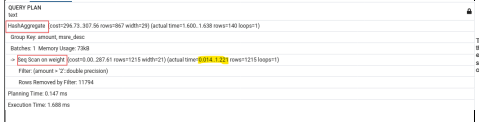
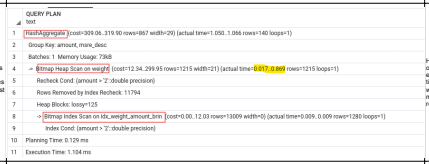
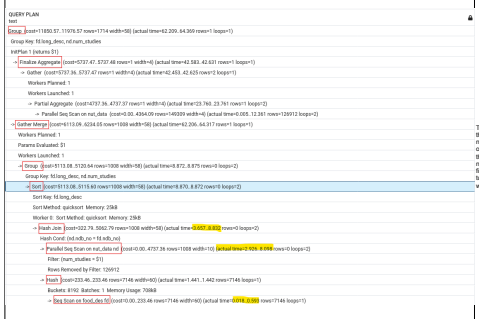
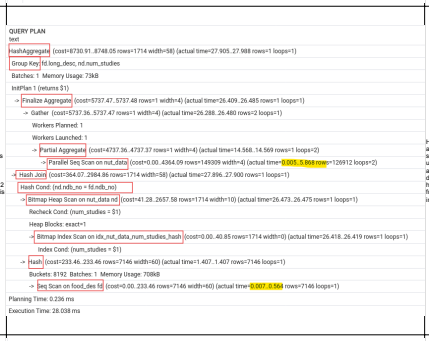
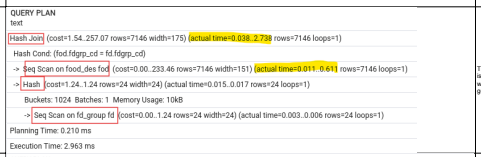
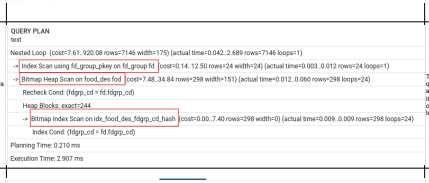
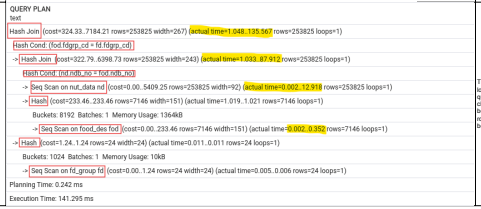
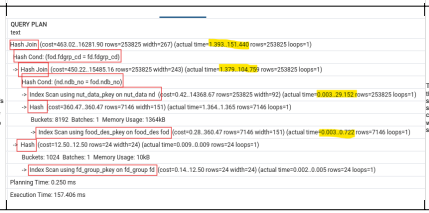
Advanced Optimization Results

		Index Information				Planning Time			Execution Time									
Query No.	Query	Variable Indexed	Type of Index Used	Index Command	Justification	Before	After	Difference	Before	After	Difference	Before	Slowest Part + Explanation	After	Slowest Part + Explanation	Physical Plan	Timing	Commands Used	Explanation
1	select * from food_das fd inner join fd_group fg on fd.fdgrp_cd = fg.fdgrp_cd where fg.fddrp_desc = 'Breakfast Cereals';	The first index (idx_fd_fdgrp_cd_btree) is created on the fdgrp_cd column in the food_das table. The second index (idx_fg_fdgrp_cd_fddrp_desc_btree) is created on two columns, fdgrp_cd and fddrp_desc, in the fd_group table	B-Tree	CREATE INDEX idx_fd_fdgrp_cd_btree ON food_das (fdgrp_cd); CREATE INDEX idx_fg_fdgrp_cd_fddrp_desc_btree ON fd_group (fdgrp_cd, fddrp_desc);	The first index helps the database quickly search for records based on the value of fdgrp_cd in that table. The second index helps the database quickly find records that match both the fdgrp_cd and fddrp_desc values in that table.	0.795	0.138	0.657	2.36	0.13	2.230		The Hash Join is the slowest part of the execution. It processed 403 iterations in total, taking 1.669 ms to complete. This operation combines rows from the food_das and fd_group tables using the condition fd.fdgrp_cd = fg.fdgrp_cd, which involves building and probing a hash table, making it more computationally intensive compared to the other steps in the plan.		The Nested Loop join is the slowest part of the execution. It processed 403 iterations in total taking 0.091 to complete.	Before indexing, the query used a hash join, relying on sequential scans for both the food_das and fd_group tables. This resulted in a planning time of 0.795 ms and an execution time of 2.360 ms. After indexing and turning off sequential scans, the query plan switched to a nested loop, utilizing index scans (idx_fd_fdgrp_cd_btree for food_das and dx_fg_fdgrp_desctbree for fd_group). This change drastically improved performance, reducing planning time to 0.138 ms and execution time to 0.130 ms by eliminating sequential scans and leveraging the indexes for efficient data access.	The use of indices reduced execution and planing time significantly.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
2	select * from weight where seq = '1';	The seq is the column indexed in the weight table.	B-Tree	CREATE INDEX idx_weight_seq_btree ON weight (seq);	This index helps the database quickly search for records based on the value of seq in the weight table.	0.258	0.177	0.081	1.39	1.165	0.225		The Sequential Scan on the weight table is the slowest part taking 1.167 ms to complete. This occurs because there's no index on the seq column, so PostgreSQL has to scan all 6,666 rows to filter for seq = '1'.		The Bitmap Heap Scan is the slowest part taking 0.731 ms as it involves fetching 6666 rows from the table using the bitmap generated by the index scan. It requires reading 124 heap blocks (as indicated in Heap Blocks: exact=124),	Before indexing, the query used a sequential scan on the 'weight' table, scanning all rows and filtering non-matching ones, with a planning time of 0.258 ms and execution time of 1.390 ms. After creating the 'idx_weight_seq_btree' index and disabling sequential scans, the query switched to a 'Bitmap Heap Scan' and 'Bitmap Index Scan', accessing only relevant rows. This reduced planning time to 0.177 ms and execution time to 1.165 ms, improving performance by leveraging the index.	The use of the index reduced execution and planing time significantly.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
3	select * from weight where seq = '1' and amount = 3;	The `seq` and The `amount` are the columns indexed in the `weight` table.	B-Tree	CREATE INDEX idx_weight_seq_amount_btree ON weight (seq, amount);	This composite index optimizes both conditions in the query, reducing scans and improving performance by directly retrieving relevant rows.	0.135	0.122	0.013	1.113	0.05	1.063		The Sequential Scan on the weight' table is the slowest part. taking 1.084 ms This occurs because there's no index on the `seq` or `amount` columns, so PostgreSQL has to scan all 12,949 rows to filter for `seq = '1'` and `amount > 3`		The Bitmap Heap Scan is the slowest part taking 0.122 ms due to the necessity of accessing table data blocks after filtering with the index.	Before indexing, the query used a sequential scan on the weight table to filter the rows where `seq = '1'` and `amount > 3`, which was costly and time consuming. However after indexing, the query preferred the physical plan we created and stopped sequentially scanning the entire weight table Instead of scanning the entire 'weight' table sequentially, the optimizer used a Bitmap Index Scan on the 'idx_weight_seq_amount_btree' index to efficiently locate matching rows, followed by a Bitmap Heap Scan to fetch the data.	The use of the index reduced execution and planing time significantly.	None	Index was used.
4	select ndb_no from weight except select ndb_no from food_des;	The ndb_no column is indexed in both the weight and food_des tables.	B-Tree	CREATE INDEX idx_weight_ndb_no_btree ON weight (ndb_no); CREATE INDEX idx_food_des_ndb_no_btree ON food_das (ndb_no);	The first index helps the database quickly search for records based on the value of 'ndb_no' in the 'weight' table, optimizing the join between 'nut_data' and 'weight'. The second index helps the database quickly search for records based on the value of 'ndb_no' in the 'datsrcln' table, optimizing queries that involve filtering or joining based on 'ndb_no' in this table.	0.063	0.135	-0.072	7.31	6.73	0.580		The Sequential Scan on the weight table is the slowest operation taking 0.894 ms due to scanning 13,009 rows.		The Append operation is the slowest operation taking 3.905 ms as it needs to process the output of both subqueries, resulting in 20,155 rows needing to be processed which is time consuming.	Before indexing, the query plan used Subquery Scans with Sequential Scans on the 'weight' and 'food_das' tables, requiring full scans of 13,009 and 7,146 rows, respectively. This was resource-intensive. After indexing, the Subquery Scans were replaced with Index Only Scans, utilizing the new indices ('idx_weight_ndb_no_btree' and 'idx_food_des_ndb_no_btree') to directly retrieve relevant rows from the indexes, reducing I/O costs and improving efficiency while maintaining the query logic.	The use of indices significantly reduce execution time, but planning time may slightly increase as the database evaluates their usefulness. This minor overhead is negligible compared to the execution time gains.	set enable_seqscan=off;	One of the indices was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
5	select ndb_no from weight except select ndb_no from datsrcln;	The ndb_no column is indexed in both the weight and datsrcln tables.	B-Tree	CREATE INDEX idx_weight_ndb_no_btree ON weight (ndb_no); CREATE INDEX idx_datsrcln_ndb_no_btree ON datsrcln (ndb_no);	These indices optimize queries that filter or compare based on the ndb_no column, allowing the database to quickly locate relevant rows without scanning the entire table	0.165	0.069	0.096	33.688	30.617	3.071		The Append operation is the slowest operation taking 19.195 ms as it needs to process the output of both subqueries, resulting in 106,854 rows needing to be processed which is time consuming.		The Append operation is the slowest operation taking 19.175 ms as it needs to process the output of both subqueries, resulting in 106,854 rows needing to be processed which is time consuming.	Before indexing, the query used sequential scans on the 'weight' and 'datsrcln' tables, processing 13,009 and 93,845 rows, respectively. The results were hashed for the EXCEPT operation, adding computational overhead and leading to a planning time of 0.165 ms and an execution time of 33.688 ms. After indexing, index-only scans replaced sequential scans, avoiding full table scans and reducing I/O costs. While hashing was still required for the EXCEPT operation, planning time dropped to 0.069 ms, and execution time improved to 30.617 ms, demonstrating enhanced efficiency.	The use of indices reduced execution and planing time significantly.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
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';	The tagname and nutr_no columns are indexed in the nutr_def table. The nutr_no and ndb_no columns are indexed in the nut_data table The ndb_no column is indexed in the weight table.	B-Tree	CREATE INDEX idx_nutr_def_tagname_nutr_no_btree ON nutr_def (tagname, nutr_no); CREATE INDEX idx_nut_data_nutr_no_ndb_no_btree ON nut_data (nutr_no, ndb_no); CREATE INDEX idx_weight_ndb_no_btree ON weight (ndb_no);	The first index helps the database quickly search for records based on the values of 'tagname' and 'nutr_no' in the 'nutr_def' table, optimizing both the filtering on 'tagname' and the join on 'nutr_no'. The second index helps the database quickly find records that match both the 'nutr_no' and 'ndb_no' values in the 'nut_data' table, optimizing the joins between 'nutr_def' and 'nut_data', as well as 'nut_data' and 'weight'. The third index helps the database quickly search for records based on the value of 'ndb_no' in the 'weight' table, optimizing the join between 'nut_data' and 'weight'.	1.004	0.785	0.219	13.839	4.311	9.528		The Hash Join is the slowest part of this physical plan as it takes 2.563 ms to process 4560 rows likely due to the large dataset and memory usage (851 KB).		The Hash Join remains the slowest part even after the help of he index, taking 3.109 ms to process 4560 rows, likely still due to the large dataset and memory usage (851 KB).	Before Indexing the planner was sequentially scanning nutr_def and weight to find the columns to hash and join. However after turning off sequential scanning the planner was forced to replace these sequential scans with index scans, which took less time.	The use of indices reduced execution and planing time significantly.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .

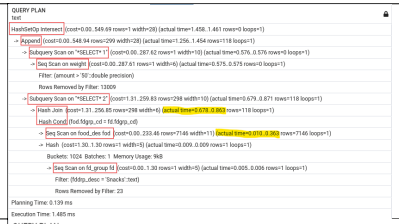
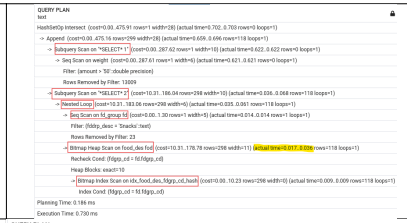
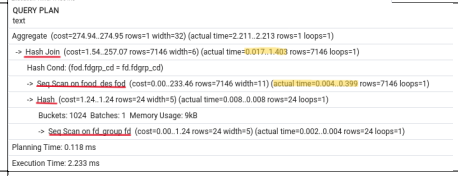
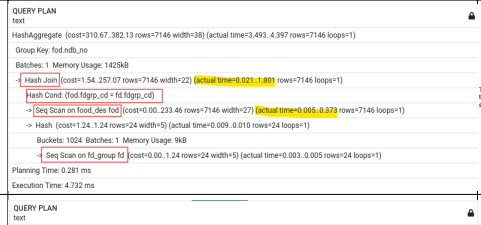
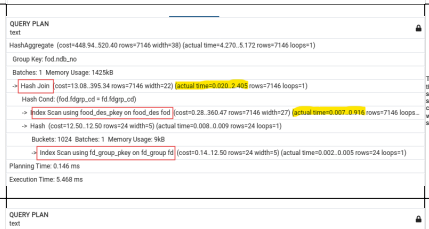
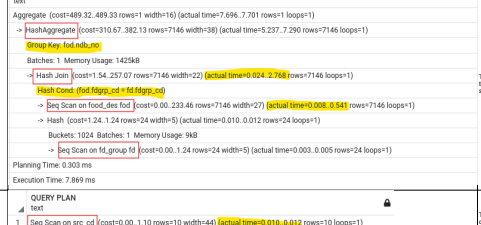
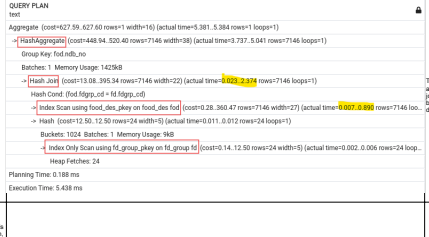
Advanced Optimization Results

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%';	The tagname and nutr_no columns are indexed in the nutr_def table. The nutr_no and ndb_no columns are indexed in the nut_data table. The ndb_no column is indexed in the weight table.	B-Tree	CREATE INDEX idx_nutr_def_tagname_nutr_no_btree ON nutr_def (tagname, nutr_no); CREATE INDEX idx_nut_data_nutr_no_ndb_no_btree ON nut_data (nutr_no, ndb_no); CREATE INDEX idx_weight_ndb_no_btree ON weight (ndb_no);	The first index helps the database quickly search for records based on the value of 'tagname' and 'nutr_no' in the 'nutr_def' table, optimizing both the filtering ('LIKE 'CA%'') and the join on 'nutr_no'. The second index helps the database quickly find records that match both the 'nutr_no' and 'ndb_no' values in the 'nut_data' table, optimizing the joins between 'nutr_def' and 'nut_data' as well as 'nut_data' and 'weight'. The third index helps the database quickly search for records based on the value of 'ndb_no' in the 'weight' table, optimizing the join between 'nut_data' and 'weight'.	0.474	0.730	-0.256	45.671	14.582	31.089	Data Output Explain Messages Notifications QUERY PLAN text 1 Hash Join (cost=6216.71..6799.84 rows=18169 width=20) (actual time=39.905..45.035 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=0.005..0.746 rows=13009 loops=1) 4 -> Hash (cost=6100.06..6100.06 rows=9332 width=18) (actual time=39.884..39.932 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=0.022..36.556 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..13.720 rows=253825 loops=1) 9 -> Hash (cost=3.70..3.70 rows=5 width=16) (actual time=0.016..0.017 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=0.006..0.013 rows=5 loops=1) 12 Filter: (tagname ~~ 'CA%'::text) 13 Rows Removed by Filter: 131 14 Planning Time: 0.474 ms 15 Execution Time: 45.671 ms	The Hash Join is the slowest part of this physical plan as it takes 5.13 ms to process 28,060 rows likely due to the large dataset and memory usage (851 KB).	Data Output Explain Messages Notifications QUERY PLAN text 1 Hash Join (cost=558.97..1338.15 rows=18169 width=20) (actual time=6.692..13.796 rows=28060 loops=1) 2 Hash Cond: (w.ndb_no = nd.ndb_no) 3 -> Index Scan using idx_weight_ndb_no_btree on weight w (cost=0.29..451.42 rows=13009 width=14) (actual time=0.007..1.966 rows=13009 loops=1) 4 -> Hash (cost=442.04..442.04 rows=9332 width=18) (actual time=6.667..6.668 rows=14486 loops=1) 5 Buckets: 16384 Batches: 1 Memory Usage: 851kB 6 -> Nested Loop (cost=0.56..442.04 rows=9332 width=18) (actual time=0.025..3.196 rows=14486 loops=1) 7 -> Index Scan using nutr_def_pkey on nutr_def nutdef (cost=0.14..15.52 rows=5 width=16) (actual time=0.007..0.030 rows=5 loops=1) 8 Filter: (tagname ~~ 'CA%'::text) 9 Rows Removed by Filter: 131 10 -> Index Only Scan using idx_nut_data_nutr_no_ndb_no_btree on nut_data nd (cost=0.42..64.97 rows=1923 width=10) (actual time=0.013..0.394 rows=2897 loops=5) 11 Index Cond: (nutr_no = nutdef.nutr_no) 12 Heap Fetches: 0 13 Planning Time: 0.730 ms 14 Execution Time: 14.582 ms	The Hash Join remains the slowest part even after the help of he index, taking 7.104 ms to process 28,060 ms, likely still due to the large dataset and memory usage (851 KB).	Before indexing the query used hash joins with sequential scans on 'weight', 'nut_data', and 'nutr_def', resulting in a planning time of 0.474 ms and an execution time of 45.671 ms. After creating indices ('idx_nutr_def_tagname_nutr_no_btree', 'idx_nut_data_nutr_no_ndb_no_btree', and 'idx_weight_ndb_no_btree') and turning off sequential scans, the query replaced sequential scans with index scans, improving efficiency. This reduced execution time to 14.582 ms (a 68% improvement) with a slight increase in planning time to 0.730 ms.	The use of indices significantly reduce execution time, but planning time may slightly increase as the database evaluates their usefulness. This minor overhead is negligible compared to the execution time gains.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
8	select count(*) from food_des where fat_factor is null;	the fat_factor column is indexed in the food_des table.	B-Tree	CREATE INDEX idx_food_des_fat_factor_null _btreeON food_des (fat_factor) WHERE fat_factor IS NULL;	We used the index to optimize the query through creating a partial index specifically for rows where 'fat_factor IS NULL'. PostgreSQL can more efficiently locate and count the matching rows without scanning the entire table. This improves the query performance by reducing the number of rows that need to be scanned and by leveraging the index structure for faster lookups.	0.068	0.109	-0.041	0.949	0.249	0.700	Data Output Explain Messages Notifications QUERY PLAN text 1 Aggregate (cost=238.24..238.25 rows=1 width=8) (actual time=0.922..0.923 rows=1 loops=1) 2 -> Seq Scan on food_des (cost=0.00..233.46 rows=1911 width=0) (actual time=0.014..0.849 rows=1911 loops=1) 3 Filter: (fat_factor IS NULL) 4 Rows Removed by Filter: 5235 5 Planning Time: 0.068 ms 6 Execution Time: 0.949 ms	The Sequential Scan processes 1911 rows from the table food_des with filtering condition where fat_factor is null which makes it the slowest operation in the physical plan taking 0.835 ms.	Data Output Explain Messages Notifications QUERY PLAN text 1 Aggregate (cost=65.72..65.73 rows=1 width=8) (actual time=0.232..0.233 rows=1 loops=1) 2 -> Index Only Scan using idx_food_des_fat_factor_null_btree on food_des (cost=0.28..60.94 rows=1911 width=0) (actual time=0.019..0.160 rows=1911 loops=1) 3 Heap Fetches: 0 4 Planning Time: 0.109 ms 5 Execution Time: 0.249 ms	The Index Scan operation is the slowest operation in the physical plan as it scans 1911 rows from food_des table where fat_factor=null which is the most time consuming operation in all the physical plan taking 0.141 ms.	Before indexing a sequential scan was performed on food_des table filtering from food_des where fat_factor is null then the count aggregate operation it does which is time and resource consuming. After indexing an index scan was performed which is a lot more optimal than a sequential scan then the count aggregate operation was performed on the result of the index also.	The use of indices significantly reduce execution time, but planning time may slightly increase as the database evaluates their usefulness. This minor overhead is negligible compared to the execution time gains.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
9	select count(*) from food_des where fat_factor is null;	The fat_factor column is indexed in the food_des table.	B-Tree	CREATE INDEX idx_food_des_fat_factor_null_btree ON food_des (fat_factor) WHERE fat_factor IS NULL;	This partial index optimizes the query by only including rows where fat_factor IS NULL, thus making the query more efficient. Instead of scanning the entire table, PostgreSQL can use this index to quickly find and count only the rows where the fat_factor column is NULL, leading to a faster query execution.	0.072	0.125	-0.053	0.908	0.465	0.443	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=0.023..0.852 rows=1911 loops=1) 2 Filter: (fat_factor IS NULL) 3 Rows Removed by Filter: 5235 4 Planning Time: 0.072 ms 5 Execution Time: 0.908 ms	The Sequential Scan is the slowest operation as it scans 1911 rows of the food_des table with filtering condition fat_factor is null which consumed 0.829 ms.	Data Output Explain Messages Notifications QUERY PLAN text 1 Bitmap Heap Scan on food_des (cost=42.31..223.42 rows=1911 width=151) (actual time=0.133..0.376 rows=1911 loops=1) 2 Recheck Cond: (fat_factor IS NULL) 3 Heap Blocks: exact=102 4 -> Bitmap Index Scan on idx_food_des_fat_factor_null_btree (cost=0.00..41.83 rows=1911 width=0) (actual time=0.102..0.102 rows=1911 loops=1) 5 Planning Time: 0.125 ms 6 Execution Time: 0.465 ms	The Bitmap Heap Scan is the slowest part of this physical plan as it involves scanning 1911 rows of the food_des table which consumed 0.133 ms even after the help of the index.	Before indexing, a sequential scan was used on the 'food_des' table to filter rows where 'fat_factor IS NULL', which was time-consuming and resource-intensive. After indexing, an optimized index scan (Bitmap Index Scan and Bitmap Heap Scan) was performed, directly locating the relevant rows and significantly improving performance. The subsequent 'MIN' aggregate operation was then executed on the filtered results from the index, making the process far more efficient.	The use of the index significantly reduce execution time, but planning time may slightly increase as the database evaluates their usefulness. This minor overhead is negligible compared to the execution time gains.	None	Index was used.
10	select min(n_factor) from food_des where fat_factor is null;	The fat_factor and n_factor columns are indexed in the food_des table.	B-Tree	CREATE INDEX idx_food_des_fat_factor_null_n_factor_btree ON food_des (fat_factor, n_factor) WHERE fat_factor IS NULL;	This partial index includes only rows where fat_factor IS NULL and covers both the filtering condition (fat_factor IS NULL) and the column used for the MIN function (n_factor). This allows PostgreSQL to efficiently find the minimum value of n_factor for the matching rows without having to scan the entire table, significantly improving query performance.	0.112	0.231	-0.119	1.003	0.292	0.711	Data Output Explain Messages Notifications QUERY PLAN text 1 Aggregate (cost=238.24..238.25 rows=1 width=8) (actual time=0.981..0.982 rows=1 loops=1) 2 -> Seq Scan on food_des (cost=0.00..233.46 rows=1911 width=8) (actual time=0.015..0.880 rows=1911 loops=1) 3 Filter: (fat_factor IS NULL) 4 Rows Removed by Filter: 5235 5 Planning Time: 0.112 ms 6 Execution Time: 1.003 ms	The Sequential Scan is the slowest operation as it scans 1911 rows of the food_des table with filtering condition fat_factor is null which consumed 0.865 ms.	Data Output Explain Messages Notifications QUERY PLAN text 1 Aggregate (cost=65.72..65.73 rows=1 width=8) (actual time=0.261..0.262 rows=1 loops=1) 2 -> Index Only Scan using idx_food_des_fat_factor_null_n_factor_btree on food_des (cost=0.28..60.94 rows=1911 width=8) (actual time=0.025..0.178 rows=1911 lo... 3 Heap Fetches: 0 4 Planning Time: 0.231 ms 5 Execution Time: 0.292 ms	The Index Scan operation is the slowest operation in the physical plan as it scans 1911 rows from food_des table where fat_factor is not null which is the most time consuming operation in all the physical plan taking 0.153 ms.	Before indexing a sequential scan was performed on food_des table filteringfrom food_des where fat_factor is null then the min aggregate operation it does which is time and resource consuming. After indexing an index scan was performed which is a lot more optimal than a sequential scan then the min aggregate operation was performed on the result of the index also.	The use of the index significantly reduce execution time, but planning time may slightly increase as the database evaluates their usefulness. This minor overhead is negligible compared to the execution time gains.	None	Index was used.
11	select sum(n_factor) from food_des where fat_factor is null;	The fat_factor and n_factor columns are indexed in the food_des table.	B-Tree	CREATE INDEX idx_food_des_fat_factor_null_n_factor_btree ON food_des (fat_factor, n_factor) WHERE fat_factor IS NULL;	This partial index includes only the rows where fat_factor IS NULL, and it also includes the n_factor column used in the SUM aggregation. This allows PostgreSQL to efficiently calculate the sum of n_factor for the rows where fat_factor IS NULL, bypassing the need to scan the entire table and improving the query performance.	0.053	0.121	-0.068	0.883	0.274	0.609	Data Output Explain Messages Notifications QUERY PLAN text 1 Aggregate (cost=238.24..238.25 rows=1 width=8) (actual time=0.862..0.863 rows=1 loops=1) 2 -> Seq Scan on food_des (cost=0.00..233.46 rows=1911 width=8) (actual time=0.014..0.773 rows=1911 loops=1) 3 Filter: (fat_factor IS NULL) 4 Rows Removed by Filter: 5235 5 Planning Time: 0.053 ms 6 Execution Time: 0.883 ms	The Sequential Scan is the slowest operation as it scans 1911 rows of the food_des table with filtering condition fat_factor is null which consumed 0.759 ms.	Data Output Explain Messages Notifications QUERY PLAN text 1 Aggregate (cost=65.72..65.73 rows=1 width=8) (actual time=0.257..0.258 rows=1 loops=1) 2 -> Index Only Scan using idx_food_des_fat_factor_null_n_factor_btree on food_des (cost=0.28..60.94 rows=1911 width=8) (actual time=0.015..0.174 rows=1911 lo... 3 Heap Fetches: 0 4 Planning Time: 0.121 ms 5 Execution Time: 0.274 ms	The Index Scan operation is the slowest operation in the physical plan as it scans 1911 rows from food_des table where fat_factor is not null which is the most time consuming operation in all the physical plan taking 0.159 ms.	Before indexing a sequential scan was performed on food_des table filteringfrom food_des where fat_factor is null then the sum aggregate operation it does which is time and resource consuming. After indexing an index scan was performed which is a lot more optimal than a sequential scan then the sum aggregate operation was performed on the result of the index also.	The use of the index significantly reduce execution time, but planning time may slightly increase as the database evaluates their usefulness. This minor overhead is negligible compared to the execution time gains.	None	Index was used.
12	select sum(n_factor) from food_des where fat_factor is not null;	The fat_factor and n_factor columns are indexed in the food_des table.	B-Tree	CREATE INDEX idx_food_des_fat_factor_not_null_n_factor_btree ON food_des (fat_factor, n_factor) WHERE fat_factor IS NOT NULL;	This partial index includes only the rows where fat_factor IS NOT NULL and covers both the filtering condition (fat_factor IS NOT NULL) and the column used in the SUM function (n_factor). This allows PostgreSQL to efficiently calculate the sum of n_factor for the rows that meet the condition, bypassing the need to scan the entire table, thus improving the query performance.	0.042	0.236	-0.194	0.963	0.717	0.246	Data Output Explain Messages Notifications QUERY PLAN text 1 Aggregate (cost=246.55..246.56 rows=1 width=8) (actual time=0.944..0.945 rows=1 loops=1) 2 -> Seq Scan on food_des (cost=0.00..233.46 rows=5235 width=8) (actual time=0.009..0.760 rows=5235 loops=1) 3 Filter: (fat_factor IS NOT NULL) 4 Rows Removed by Filter: 1911 5 Planning Time: 0.042 ms 6 Execution Time: 0.963 ms	The Sequential Scan is the slowest operation as it scans 5235 rows of the food_des table with filtering condition fat_factor is not null which consumed 0.751 ms.	Data Output Explain Messages Notifications QUERY PLAN text 1 Aggregate (cost=183.90..183.91 rows=1 width=8) (actual time=0.699..0.700 rows=1 loops=1) 2 -> Index Only Scan using idx_food_des_fat_factor_not_null_n_factor_btree on food_des (cost=0.28..170.81 rows=5235 width=8) (actual time=0.017..0.461 rows=5235 loops=1) 3 Heap Fetches: 0 4 Planning Time: 0.206 ms 5 Execution Time: 0.717 ms	The Index Scan operation is the slowest operation in the physical plan as it scans 5235 rows from food_des table where fat_factor is not null which is the most time consuming operation in all the physical plan taking 0.444 ms.	Before indexing a sequential scan was performed on food_des table filtering where fat_factor is not null then it performs the sum aggregate operation it does which is time and resource consuming. However after use of the index an index scan was performed which is a lot more optimal than a sequential scan then the sum aggregate operation was performed on the result of the index also.	The use of the index significantly reduce execution time, but planning time may slightly increase as the database evaluates their usefulness. This minor overhead is negligible compared to the execution time gains.	None	Index was used.
13	select tagname from nutr_def nutdef inner join datarcdn dal on dal.nutr_no = nutdef.nutr_no inner join data_src ds on ds.datasrc_id = dal.datasrc_id where ds.vol_city = 'Cincinnati';	datarcdn(datasrc_id)	btree	create index idx_dal_datasrcid_btree on datarcdn (datasrc_id);	datasrc_id is already a primary key in data_src (ds) but not in datarcdn (dal) so we added an index with the hopes of encouraging an index join instead of multiple nested loops	0.704	0.366	0.338	2.821	0.212	2.609	QUERY PLAN text Nested Loop (cost=0.71..2649.38 rows=256 width=5) (actual time=0.252..2.794 rows=27 loops=1) -> Nested Loop (cost=0.56..2607.77 rows=256 width=4) (actual time=0.249..2.765 rows=27 loops=1) -> Index Scan using data_src_pkey on data_src ds (cost=0.15..25.55 rows=1 width=7) (actual time=0.032..0.047 rows=1 loops=1) Filter: (vol_city = 'Cincinnati'::text) Rows Removed by Filter: 368 -> Index Only Scan using datarcdn_pkey on datarcdn dal (cost=0.42..2579.24 rows=298 width=11) (actual time=0.215..2.713 rows=27 loops=1) Index Cond: (datasrc_id = ds.datasrc_id) Heap Fetches: 0 -> Index Scan using nutr_def_pkey on nutr_def nutdef (cost=0.14..0.16 rows=1 width=9) (actual time=0.001..0.001 rows=1 loops=27) Index Cond: (nutr_no = dal.nutr_no) Planning Time: 0.704 ms Execution Time: 2.821 ms	The slowest part of this Query is the Index Only Scan using the datasrcdn_pkey taking nearly 2.5 ms to execute. While this can be weird, sometimes Index Only Scans will still need to access memory and given that it had to go through 27 rows this could be why it took so long.	QUERY PLAN text Hash Join (cost=25.63..599.16 rows=256 width=5) (actual time=0.137..0.183 rows=27 loops=1) Hash Cond: (dal.nutr_no = nutdef.nutr_no) -> Nested Loop (cost=0.15..541.58 rows=256 width=4) (actual time=0.084..0.125 rows=27 loops=1) -> Index Scan using data_src_pkey on data_src ds (cost=0.15..25.55 rows=1 width=7) (actual time=0.043..0.059 rows=1 loops=1) Filter: (vol_city = 'Cincinnati'::text) Rows Removed by Filter: 365 -> Bitmap Heap Scan on datarcdn dal (cost=4.60..513.05 rows=298 width=11) (actual time=0.018..0.041 rows=27 loops=1) Recheck Cond: (datasrc_id = ds.datasrc_id) Heap Blocks: exact=20 -> Bitmap Index Scan on idx_dal_datasrcid_btree (cost=0.00..4.53 rows=298 width=0) (actual time=0.012..0.012 rows=27 loops=1) Index Cond: (datasrc_id = ds.datasrc_id) -> Hash (cost=15.16..15.16 rows=136 width=9) (actual time=0.045..0.046 rows=136 loops=1) Buckets: 1024 Batches: 1 Memory Usage: 14kB -> Index Scan using nutr_def_pkey on nutr_def nutdef (cost=0.14..15.16 rows=136 width=9) (actual time=0.004..0.025 rows=136 loops=1) Planning Time: 0.366 ms Execution Time: 0.212 ms	There isn't a specific part that takes an absurdly long time in it's own right which suggests that this is a good physical plan for the query specifically. However more memory is being used to account for the use of indices, this could present another array of problems but not temporal ones.	After forcing the planner not to use sequential scans, the nested loops and sequential scans were changed out for Hash Joins, Bitmap Index Scans, Bitmap Heap Scans and Index Scans. It was able to make use of both existing indices and the index we created.	The change from Iterative methods to Index Based Algorithms significantly reduced execution time.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .

Advanced Optimization Results

14	select * from weight where amount > 2;	weight(amount)	brin	create index idx_weight_amount_brin on weight using BRIN (amount);	the filtering condition made us think of BRIN indices and we hoped they would help the planner skip chunks of data especially as it was not a between range query but a less than one.	0.052	0.366	-0.314	1.123	0.212	0.911		There's only one operation in this query which is a sequential scan to filter the elements. This makes sense initially because of the simplicity of the filtering condition.		There isn't a specific part that stands out as taking super long which suggests that this is a good physical plan for the query specifically. However more memory is being used to account for the use of indices, this could present another array of problems but not temporal ones.	After forcing the planner not to use sequential scans, the nested loops and sequential scans were changed out for Hash Joins, Bitmap Index Scans, Bitmap Heap Scans and Index Scans. It was able to make use of both existing indices and the index we created.	The change from Iterative methods to Index Based Algorithms significantly reduced execution time. This could have been also due to nearly 10% of the table being removed. A BRIN Index would have skipped over the 1384 tuples giving way for faster execution.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
15	select * from weight where amount > 2;	weight(amount)	brin	create index idx_weight_amount_brin on weight using BRIN (amount);	the filtering condition made us think of BRIN indices and we hoped they would help the planner skip chunks of data especially as it was not a between range query but a greater than one.	0.049	0.057	-0.01	0.845	0.964	-0.12		There's only one operation in this query which is a sequential scan to filter the elements. This makes sense initially because of the simplicity of the filtering condition.		Here the Bitmap heap scan on weight is the longest part even after removing it's child time from the execution. This would make sense because most of the data is being removed by the filter.	Initially the planner still went for a sequential scan so after forcing it to try other methods it went for a bitmap Index Scan with our created index to filter and a bitmap heap scan on weight after to retrieve the rest of the columns.	The changes this time did not result in significant changes and actually worsened execution. This makes sense when we look at the number of rows being filtered out. Most of the data is greater than so it does make sense to retrieve sequentially instead of using multiple checks for a BRIN Index.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
16	select msrc_desc, avg (amount) from weight where amount > 2 group by amount, msrc_desc;	weight(amount)	brin	create index idx_weight_amount_brin on weight using BRIN (amount);	the filtering condition made us think of BRIN indices and we hoped they would help the planner skip chunks of data especially as it was not a between range query but a less than one.	0.147	0.057	0.090	1.688	1.176	0.512		The sequential scan on weight is the longest part of the query execution time wise which makes sense because its removing most of the rows.		Here the Bitmap heap scan on weight is the longest part even after removing it's child time from the execution. This would make sense because most of the data is being removed by the filter.	Initially the planner still went for a sequential scan so after forcing it to try other methods it went for a bitmap Index Scan with our created index to filter and a bitmap heap scan on weight after to retrieve the rest of the columns.	The changes this time did not result in significant changes and actually worsened execution. This makes sense when we look at the number of rows being filtered out. Most of the data is greater than so it does make sense to retrieve sequentially instead of using multiple checks for a BRIN Index.	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
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);	nut_data(num_studies)	hash	create index idx_nut_data_num_studies_hash on nut_data using hash (num_studies);	By creating a hash index on num studies we were hoping to help the planner filter by the max value faster	0.331	0.236	0.095	73.37	28.038	45.332		The longest part of this query is the parallel sequential scan on nut_data which takes nearly time on its own. It's used as part of the filtering process on num_studies and if we look the filtering process removes 129912 tuples so it makes sense why this would take the longest times.		Here the longest part is once again a parallel sequential scan on nut_data that was use as part of the partial aggregate. This was probably due to the volume of the rows however it decreased slightly from the one without the index.	After introducing the hash index the planner started preferring bitmap index scans and Hash Algorithms more generally. This suggests that the index was a good idea and this makes sense because it's a direct solution to the bottleneck that was occuring with the filtering.	The introduction of the index had a positive impact on execution time because of the use of Hash Algorithms instead of their iterative counterparts.	none	Index was used
18	select * from fd_group fd inner join food_des fod on fd.fdgrp_cd = fod.fdgrp_cd;	food_des(fdgrp_cd)	hash	create index idx_food_des_fdgrp_cd_hash on food_des using hash(fdgrp_cd);	This index was made to decreaseleliminate the hashing on the fly time and that way speeding the query up	0.210	0.210	0.000	2.963	2.907	0.056		The sequential scan on food_des is the longest part in this query which is probably because its going through 7146 rows.		The longest operation in this query is now the nested loop and thats probably because it's internal children operations are operating on less tuples	Because the hashing was already handled by our index, the planner opted for a nested loop with bitmap heap and index scans. However the overarching structure is the same	Time was unaffected by the swapping of Iterative and hashing algorithms	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
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;	food_des(fdgrp_cd)	hash	create index idx_food_des_fdgrp_cd_hash on food_des using hash(fdgrp_cd);	This index was made to decreaseleliminate the hashing on the fly time and that way speeding the query up	0.242	0.250	-0.008	141.295	157.406	-16.111		The hash join is by far the longest single operation in this query, even after factoring out it's children operations. This is because of the sheer number of rows it was matching, 253825 to be exact.		The slowest part again was the hash join and this makes sense because the actual structure of the query wasn't changed, sequential scans were just substitued for index scans	Simply, the sequential scans became Index scans on the indices already created by Postgres. This is because most of the attributes we are dealing with are primary keys so it would make sense for the planner to choose them over a user created index on another attribute.	Timing wasn't significantly changed by the use of index scans instead of sequential scans	set enable_seqscan=off;	No matter what we did the index would not be used. This suggests that the index was not useful at all to the query and the planner preffered other indices.

Advanced Optimization Results

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');	food_des(fdgrp_cd)	hash	create index idx_food_des_fdgrp_cd_hash on food_des using hash(fdgrp_cd);	This index was made to decreaseie/eliminate the hashing on the fly time and that way speeding the query up	0.139	0.186	-0.047	1.485	0.730	0.755		The longest single elements of this query are hash join and sequential scans on food_des		The longest single operation in this query plan is bitmap heap scan on food_des. This was probably because of it going through 118 rows.	The only change that happened was that the sequential scan on food_des became a bitmap index scan	Timing significantly changed by the use of bitmap index scans instead of sequential scans	none	Index was used
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;	food_des(fdgrp_cd)	hash	create index idx_food_des_fdgrp_cd_hash on food_des using hash(fdgrp_cd);	This index was made to decreaseie/eliminate the hashing on the fly time and that way speeding the query up	0.118	0.135	-0.02	2.233	1.898	0.335		The longest single elements of this query are hash join and sequential scans on food_des		The longest operation in this query is now the nested loop and thats probably because it's internal children operations are operating on less tuples	Because the hashing was already handled by our index, the planner opted for a nested loop with bitmap heap and index scans. However the overarching structure is the same	Time was slightly affected by the swapping of iterative and hashing algorithms	set enable_seqscan=off;	Index was not used so we had to turn off the sequential scan to force the query planner to utilize the index .
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;	food_des(fdgrp_cd)	hash	create index idx_food_des_fdgrp_cd_hash on food_des using hash(fdgrp_cd);	This index was made to decreaseie/eliminate the hashing on the fly time and that way speeding the query up	0.281	0.146	0.135	4.732	5.468	-0.736		The longest single elements of this query are hash join and sequential scans on food_des		The slowest part again was the hash join and this makes sense because the actual structure of the query wasn't changed, sequential scans were just substituted for index scans	Simply, the sequential scans became Index scans on the indices already created by Postgres. This is because most of the attributes we are dealing with are primary keys so it would make sense for the planner to choose them over a user created index on another attribute.	Timing wasn't significantly changed by the use of index scans instead of sequential scans	set enable_seqscan=off; set enable_indexonlyscan = off;	No matter what we did the index would not be used. This suggests that the index was not useful at all to the query and the planner preffered other indices.
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;	1. food_des(fdgrp_cd) 2. food_des(fat_factor,pro_factor,ndb_no)	Btree	create index idx_food_des_fdgrp_cd_hash on food_des using hash(fdgrp_cd); create index idx_food_des_fat_factor_pro_factor_ndb_no_cd_btree on food_des (fat_factor, pro_factor, ndb_no);	The first index was done to make the innerjoin in the subquery easier by allowing for a faster hash join instead of hashing on the fly, while the second index was made with the intention of speeding up the group by	0.303	0.188	0.115	7.869	5.438	2.431		The longest single elements of this query are hash join and sequential scans on food_des		The longest operations again are the index scans and hash joins and this makes sense because the actual structure did not change	Simply, the sequential scans became Index scans on the indices already created by Postgres. This is because most of the attributes we are dealing with are primary keys so it would make sense for the planner to choose them over a user created index on another attribute.	Timing slightly changed by the use of index scans instead of sequential scans	set enable_seqscan = off;	No matter what we did the index would not be used. This suggests that the index was not useful at all to the query and the planner preffered other indices.
24	select srccd_desc from src_cd;	None	None	None	This will select all values of that column adding an index will only increase complexity	0.043	0.043	0.00	0.024	0.024	0.00		There's only one operation in this query which is a sequential scan, which makes sense because we're just projecting			This will select all values of that column adding an index will only increase complexity		nothing	There's no use of trying to do something that isn't a sequential scan