			Index Information			
Query No.	Assigned to	Query	Variable Indexed	Type of Index Used	Index Command	Justification
1	Nour	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl. ndb_no = nd.ndb_no and dl. nutr_no = nd.nutr_no;	nut_data (ndb_no, nutr_no)	B-tree	CREATE INDEX idx_ndb_no_nutr_no_btree ON nut_data (ndb_no,nutr_no);	we created this index to make finding matching data in nut_data faster by organizing the data in a way that lets the database skip over irrelevant rows when performing the join.
2	Nour	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl. ndb_no = nd.ndb_no and dl. nutr_no = nd.nutr_no where year > 2000 AND year < 2001;	datsrcln (datasrc_id)	B-tree	CREATE INDEX idx_datasrc_id_btree ON datsrcln (datasrc_id);	we created this index on datasrc_id to speed up joins between the data_src and datsrcln tables. By indexing datasrc_id, the database can quickly locate matching rows without scanning the entire datsrcln table, which significantly improves query performance when joining on this column.
3	Nour	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl. ndb_no = nd.ndb_no and dl. nutr_no = nd.nutr_no where year > 2000;	datsrcln (datasrc_id, ndb_no, nutr_no)	B-tree	CREATE INDEX idx_datsrcln_datasrc_id_ndb_no_nutr_no_btree ON datsrcln (datasrc_id, ndb_no, nutr_no);	we created this index to optimize the query by speeding up the join operations between the `datsrcln`, `data_src`, and `nut_data` tables. The index on the columns `datasrc_id`, `ndb_no`, and `nutr_no` helps efficiently locate matching rows during the join process. Specifically, it speeds up the innerjoin between `datsrcln` and `data_src` on `datasrc_id`, and the innerjoin between `datsrcln` and `nut_data` on `ndb_no` and `nutr_no`. By creating a composite index on these columns, the database can quickly find the relevant rows in the `datsrcln` table, improving the performance of the join operation and reducing the amount of data scanned.
4	Nour	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl. ndb_no = nd.ndb_no and dl. nutr_no = nd.nutr_no where min is not null;	datsrcln (datasrc_id, ndb_no, nutr_no)	B-tree	CREATE INDEX idx_datasrc_id_ndb_no_nutr_no_btree ON datsrcln (datasrc_id, ndb_no, nutr_no);	we created this index to optimize the query by speeding up the JOIN operations between the `data_src`, `datsrcln`, and `nut_data` tables. This composite index on the columns `datasrc_id`, `ndb_no`, and `nutr_no` helps efficiently locate matching rows during the join between `datsrcln` and the `data_src` table on `datasrc_id`, and the join between `datsrcln` and `nut_data` on `ndb_no` and `nutr_no`. Additionally, this index reduces the amount of data scanned by helping the database quickly identify the relevant rows in the `datsrcln` table before the filtering condition `min IS NOT NULL` is applied. By optimizing the join process, this index contributes to improved query performance.
5	Nour	SELECT * FROM data_src ds inner join datsrcln dl on ds.datasrc_id = dl.datasrc_id inner join nut_data nd on dl. ndb_no = nd.ndb_no and dl. nutr_no = nd.nutr_no where min is null;	nut_data (min)	B-tree	CREATE INDEX idx_min_btree ON nut_data (min);	we created this index to optimize the query by speeding up the filtering process in the `WHERE min IS NULL` condition. The index on the `min` column helps the database quickly locate rows where `min` is `NULL`, avoiding a full table scan of the `nut_data` table. By creating an index specifically on the `min` column, the database can efficiently search for rows that meet this condition, improving the overall query performance. This index reduces the number of rows the system needs to process during the filtering phase, ultimately making the query faster.
6	Nour	select * from nutr_def nu inner join nut_data nd on nu. nutr_no = nd.nutr_no where max is null;	nut_data (nutr_no, max)	B-tree	CREATE INDEX idx_nutr_no_max_btree ON nut_data (nutr_no, max);	we created this index to optimize the query by improving the performance of the JOIN operation on nutr_no and speeding up the filtering condition max IS NULL. The database can quickly find rows where max is NULL and perform the join on nutr_no efficiently.

	Index Statistics			Query Plan Time			Query Execution Time			Latency Comparison (ms)			TPS Comparison		
Query No.	Index Scan	Index Tuple Read	Index Fetch	Before Index	After Index	Difference	Before Index	After Index	Difference	Before Index	After Index	Difference	Before Index	After Index	Difference (%)
1	3000	253668000	253666000	8706.250789	988.926857	-7717.323932	3521683.368	279186.1843	-3242497.184	616.973	287.579	-329.394	1.620822	3.477349	114.5423125
2	4000	4000	0	1122.853346	810.848923	-312.004423	65.536525	35.999425	-29.5371	1.628	1.301	-0.327	615.505003	769.913917	25.08654085
3	91000	43838000	0	1936.423345	1628.012117	-308.411228	440350.498	439662.6968	-687.801295	323.691	274.771	-48.92	3.089404	3.639433	17.80372525
4	4000	4000	0	1433.386017	1038.364735	-395.021282	316209.0617	145189.3513	-171019.7103	321.497	150.137	-171.36	3.110475	6.660684	114.1371977
5	0	0	0	964.43531	1030.091864	65.656554	238024.4579	4593.316069	-233431.1419	243.511	258.644	15.133	4.106641	3.866354	-5.85118105
6	2000	2000	0	268.693191	1305.808535	1037.115344	1079405.252	5435.00484	-1073970.248	1094.35	322.832	-771.518	0.913787	3.097605	238.9854528

	Behaviour Explanation		
Query No.	Latency	TPS	Overall
1	The query execution time reduced to 279186.1843 ms, meaning the time taken to process each query decreased significantly. This reduction in execution time directly translates into lower latency because the system can respond to each query much faster.	The query plan time dropped to 988.926857 ms, and the execution time reduced to 279186.1843 ms. This dramatic reduction in time means that the database can now handle queries more efficiently, allowing it to process more queries in the same period. This increase in processing efficiency results in an increase in TPS, as more transactions can be completed in a shorter time frame.	the index was successful in reducing the query execution time because it helped speed up the join operation. The index allowed the database engine to locate the tuples faster, improving the overall query plan and reducing the time spent on looking up the relevant rows. This was reflected in the idx_scan (index scans) and idx_tup_read (index tuple reads) counts, which suggest that the index facilitated faster access to data, even if it wasn't directly fetching the tuples for every operation.
2	Execution time dropped by approximately 29,537.1 ms (from 65,536.525 ms to 35,999.425 ms), meaning the query turnaround time was significantly shortened, reducing latency	TPS increased from 615.500603 to 769.913197, showing an improvement of 25.0658485%, indicating that more transactions could be processed per second due to the decreased execution time.	The index was effective, as it optimized the join process by making tuple access more efficient, thus significantly reducing execution time
3	Latency decreased from 323.691 ms to 274.771 ms, reducing by 48.92 ms. This reduction means that the time taken for the database to respond to the query decreased slightly	TPS increased from 3.089404 to 3.639433, showing an improvement of approximately 17.803265%. This indicates that the index allowed more transactions to be processed per second, albeit with a moderate impact	The index was moderately effective although the overall execution time and latency showed some improvement, the effect was limited due to the query's complexity and data volume. The primary improvement came from optimizing the query plan time, which helped the database locate and join tuples more efficiently
4	Latency decreased from 321.497 ms to 150.137 ms (a reduction of 171.36 ms). The latency reduction indicates that the time required for the database to respond to the query was significantly shortened. This is a direct result of the index making data access faster and reducing wait times	TPS increased from 3.110475 to 6.660684, showing a 114.15% increase. The index's impact on execution time allowed the database to complete the query faster, enabling more transactions per second. This improvement in TPS reflects the database's enhanced efficiency due to the index, as it could process twice as many transactions in the same time span.	The index was highly effective in optimizing the query's performance. By reducing both the query plan time and execution time, it minimized the response delay (lowering latency) and significantly boosted the number of transactions processed per second (TPS). The index's usage is evident from the Index Scan and Index Tuple Read values, confirming that it helped the database engine access relevant rows faster and reduce I/O operations.
5	The latency has increased from 243.511 ms to 258.644 ms. This suggests that after the index was created, the query execution time increased. However, since the index wasn't actually used (as seen from the index scan stats), the increased latency might be due to factors such as additional overhead for managing and maintaining the index, even if it wasn't leveraged. This could also be due to naturally occuring differences in resources during the time of execution.	The transactions per second (TPS) have decreased slightly from 4.106641 to 3.866354. The decrease in TPS further suggests that the system became slightly less efficient after the index creation. Again, this is likely due to the index not being used, which can result in an increase in unnecessary overhead or resource contention (disk I/O, CPU cycles), even though the query performance itself didn't benefit from the index.	The index was unsuccessful because the query planner didn't use it. This could be due to the min column having many NULL values, making the index less selective. As a result, the planner likely opted for a sequential scan or another execution method, which it deemed more efficient. Additionally, the overhead of maintaining the index could have contributed to the observed performance issues without providing any actual benefit for the query.
6	The latency reduction aligns with the performance improvements in terms of query execution time and TPS, indicating that the index helped reduce the time spent fetching and processing rows.	TPS went up from 0.91 before the index to 3.10 after it. This increase suggests that the index allows the system to handle more queries or data operations per second. The database can process more requests because the time per query is reduced, likely due to more efficient access to relevant rows.	The index was likely used in some form to optimize the query. Although the Index Fetch = 0 suggests that the index might not have been fully used for fetching rows, the overall performance improvements (reduced query execution time, reduced latency, and increased TPS) suggest that the index played a key role in improving the query's performance, especially for accessing the rows where the max column is NULL.

7	Nour	select * from nutr_def nu inner join nut_data nd on nu. nutr_no = nd.nutr_no where max is null order by max;	nut_data (nutr_no, max)	B-tree	CREATE INDEX idx_nutr_no_max_btree ON nut_data (nutr_no, max);	we created this index to optimize both the JOIN operation on nutr_no and the ORDER BY max clause. It allows the database to efficiently match rows between the nutr_def and nut_data tables based on nutr_no and also speeds up sorting by the max column. Additionally, because of the WHERE max IS NULL condition, the database can quickly find the rows where max is NULL and then use the index to perform the sorting.	2000	2000	0	322.729965	273.86222	-48.867745	1247784.885	477501.8147	-770283.0708	1268.045	491.719	-776.326	0.788618	2.033692	157.8804947	The latency decreased from 1268.05 ms to 491.72 ms. This is a significant reduction, which indicates that the query now executes faster overall with the new index. The reason for this could be that the index helps in quickly narrowing down the rows in the nut_data table that are relevant to the query's conditions, specifically the max IS NULL filter. Since the index is likely being used to directly access rows where the nutr_no and max values match the query conditions, the time to process the query is reduced.	The TPS increased from 0.79 TPS to 2.03 TPS, which is a clear indication of improved performance. A higher TPS means more queries can be processed in the same amount of time. This increase is likely due to the reduction in latency, as fewer resources are required to execute the query after the index creation. The index is optimizing access to the nut_data table, improving the overall throughput of the system by reducing time spent on this specific query.	The index was indeed used in the query execution process, as evidenced by the reported index scans and index tuple reads, both of which remained constant at 2000 before and after the index was created. This indicates that the index is actively being used to retrieve rows, although the index fetch count remained at 0, suggesting that the database did not need to perform additional lookups outside of what was retrieved by the index.
8	Nour	SELECT food_des.ndb_no, food_des.long_desc, gm_wgt FROM food_des INNER JOIN weight ON weight.ndb_no=food_des. ndb_no where weight.msre_desc ='serving';	weight (ndb_no, msre_desc)	B-tree	CREATE INDEX idx_weight_ndb_no_msre_desc_btree ON weight (ndb_no, msre_desc);	we created this index to help optimize the JOIN by speeding up the JOIN operation between the food_des and weight tables also including msre_desc in the composite index improves the filtering performance of the WHERE condition (weight. msre_desc = 'serving') by reducing the number of rows scanned.	2000	2000	0	438.668277	195.147978	-243.520299	6564.345575	3058.491256	-3505.854319	7.732	3.79	-3.942	129.398546	264.091859	104.0918288	the database can quickly locate the relevant rows, especially for the JOIN and WHERE operations. Instead of scanning through many rows to find matches, the index provides a direct path to the data, reducing the time it takes to process the query	with the index the database performs fewer I/O operations and reads less data from the disk, allowing it to handle more queries in the same amount of time. This efficiency boost leads to a double in TPS since each query takes less time to complete.	The index was effectively used by the query. It significantly reduced the time for query planning, execution, and retrieval of results, as well as improved system throughput (TPS) and reduced latency. The combination of these factors indicates that the index had a highly positive impact on query performance.
9	Nour	select gm_wgt from weight;	None	None	None	We don't need to use an index for this query because it retrieves all rows from the gm_wgt column without any filtering.A sequential scan is more efficient than using an index in this case.	0	0	0	69.380863	69.380863	0	8624.459237	8624.459237	0	9.267	9.267	0	107.946021	107.946021	0	No changes	No changes	No index was used as there is no filtering condition applied to this query therefore a sequential scan is more efficient .
10	Nour	select min(gm_wgt) from weight;	weight (gm_wgt)	B-tree	CREATE INDEX idx_gm_wgt_btree ON weight (gm_wgt);	B+ Trees are good for aggregates and especially ordered queries, since we are looking for the minimum value, a tree makes sense	1000	1000	0	52.274247	39.031339	-13.242908	1094.417213	15.067676	-1079.349537	1.469	0.395	-1.074	682.185236	2544.880566	273.0483206	After the index was created, the latency dropped significantly, which is consistent with the reduced query execution time. A more efficient plan, facilitated by the index, reduces the time the database takes to process the query and send back the result.	After creating the index, the TPS increased significantly. This is likely due to the reduction in query execution time, which frees up system resources to handle more transactions overall. With faster query execution, the database engine is able to process a higher volume of queries.	The index was used effectively for the query. The reduction in execution time, query planning time, latency, and the increase in TPS all indicate that the index on gm_wgt significantly optimized the query. The database engine could leverage the index to efficiently fetch the minimum value without scanning the entire table.
11	Nour	select min(gm_wgt), msre_desc from food_des INNER JOIN weight ON weight.ndb_no=food_des. ndb_no group by msre_desc;	weight (ndb_no, msre_desc)	B+ Tree	CREATE INDEX idx_weight_ndb_no_msre_desc_btree ON weight (ndb_no, msre_desc);	we created this index to help optimize the JOIN by speeding up the JOIN operation between the `food_des` and `weight` tables through the `ndb_no` column. Including `msre_desc` in the composite index improves the performance of the group by operation, as it allows the database to efficiently group the results by `msre_desc` without scanning the entire table. This way, both the join and the grouping are handled faster, making the query more efficient.	2000	2000	0	1920.919013	205.115698	-1715.803315	58675.11753	6459.753251	-52215.36428	8.004	7.29	-0.714	124.989904	137.22109	9.785739175	The decrease in latency (from 8.004 ms to 7.29 ms) is modest, but still shows an improvement. This indicates that the time required for each individual operation in the query has decreased slightly, likely because the index enables faster row retrieval and join operations. However, the change is small because the query might still involve other overheads (like network or disk I/O), not just index lookups.	The increase in TPS (from 124.99 to 137.22) indicates that the system is now able to handle more transactions per second, which aligns with the improved efficiency brought by the index. More TPS typically reflects better performance in handling multiple queries or concurrent workloads, as the query has become faster and more resource-efficient.	The index was used. The query plan and execution times indicate that the index was somewhat utilized. The absence of `Index Fetch` suggests that the index alone was sufficient to retrieve the required data, improving query efficiency. The execution time dropped significantly from 58.7 seconds to 6.46 seconds, indicating a substantial performance improvement. This shows that the index effectively sped up the query, making it a good optimization. The small reduction in latency and increase in TPS further support that the index had a positive impact.
12	Danya	select count(*) from (select min(gm_wgt), msre_desc from food_des INNER JOIN weight ON weight.ndb_no=food_des. ndb_no group by msre_desc) as t;	Weight(ndb_no)	B+ Tree	CREATE INDEX idx_weight_ndb_no_btree ON weight (ndb_no);	In the Query Execution Plan it showed that 13000 rows were expected in a scan on this row so I chose it to be indexed and I used a b+ because we were looking for an exact value match and B+ trees support it	22000	22000	0	4878.220493	2359.528821	-2518.691672	128572.7105	59022.52301	-69550.18745	14.564	6.178	-8.386	68.666073	161.876377	135.7443348	Execution time dropped by 70000ms meaning that the turnaround between sending in the query and getting a response back was a lot shorter, hence less latency	The index caused for less I/O operations, allowing for faster completion of the query. This meant that more can be done per second thus increasing TPS	The index was successful in reducing the query execution time because it sped up the join, not because actual tuples were fetched from it, it helped the engine locate tuples faster and this was shown in the idx_scan and idx_tup_read counts
13	Danya	select min(gm_wgt), msre_desc from weight where gm_wgt > 100 group by msre_desc;	weight (msre_desc)	B+ Tree	CREATE INDEX idx_weight_msre_desc_btree ON weight (msre_desc);	In the query we group by msre_desc, so we created an index on this column to speed up the grouping, we used a B+ Tree because B+ trees work better for clustering than the other non geospatial indices in PostgreSql	0	0	0	1132.428886	542.108149	-590.320737	46565.70391	20836.19384	-25729.51007	5.21	2.515	-2.695	191.959918	397.697222	107.1772202	Since the index wasn't used we can't tell the exact reason for the changes, htey likely had to do with computer resources and were minimal	Since the index wasn't used we can't tell the exact reason for the changes, htey likely had to do with computer resources and were minimal	This index was unsuccessful as it was not chosen by the engine, instead a sequential scan was selected. This could have to do with the >100 condition as the resultant set is most of the table therefore the use of the index could not have been jsutified
14	Danya	select gm_wgt from weight where gm_wgt > 100;	weight(gm_wgt)	BRIN	CREATE INDEX idx_weight_gm_wgt_brin ON weight USING BRIN (gm_wgt);	This query is a range query and BRIN Indexes are good for Range Queries on large datasets because it allows us to skip large portions of the table at a time	0	0	0	721	475.739124	-245	43042.28297	20881.47605	-22160.80692	4.838	2.569	-2.269	206.741878	389.362056	88.3324558	The Latency change was due to reasons that are not the index	It was a really simple query so adding an index wasn't seen as useful	Overall, the engine decided a sequential scan was more useful than trying to use our index, this could have been because it was retrieving a large portion of the table so it didn't make sense to use an index to do so
15	Danya	select * from src_cd sc inner join nut_data nd on sc.src_cd = nd.src_cd inner join food_des fd on fd.ndb_no = nd.ndb_no where sc.src_cd = 2;	src_cd(src_cd)	Hash	CREATE INDEX idx_src_cd_src_cd_hash ON src_cd USING HASH (src_cd);	Hash indicies are faster then B+ Trees for equality searches, which is done in this query (sc. src_cd = 2)	0	0	0	2753.067123	1650.204178	-1102.862945	145.369887	81.225261	-64.144626	0.622	0.526	-0.096	1609.584032	1902.606521	18.2048581	The Latency change was minimal because the index wasn't necessary nor used so it didn't really affect request/response time	TPS increased by nearly 18% because more resources wereavailable which had very little to do with the index	This index was unsuccessful as it was not chosen by the engine, instead a sequential scan was selected. This could have to do with the sc.src_cd = 2 condition as the resultant set is a large enough portion of the table to invalidate the use of the index.
16	Danya	select * from src_cd sc inner join nut_data nd on sc.src_cd = nd.src_cd inner join food_des fd on fd.ndb_no = nd.ndb_no;	nut_data (src_cd, ndb_no)	B+ Tree	CREATE INDEX idx_nut_data_src_cd_ndb_no_btree ON nut_data (src_cd, ndb_no);	These columns are used in the different joins, so we thought it would be a good idea to index them to speed up the joins. Moreover, the other columns are primary keys of their tables therfore are already indexed	0	0	0	383.037319	381.996477	-1.040842	1373365.742	613378.3068	-759987.4347	1401.52	634.479	-767.041	0.713512	1.576102	120.8935519	The Latency change was due to reasons that are not the index	Eventhough TPS increased by 120% , the index wasn't used meaning it probably had more to do with the laptop's available resources	The Index wasn't used, suggesting that these columns being indexed were not advantageous enough to be used. nut_data is a large table and most of it is being retrieved therefore it makes more sense for the engine to just preform a sequential scan.
17	Danya	select * from src_cd sc inner join nut_data nd on sc.src_cd = nd.src_cd inner join food_des fd on fd.ndb_no = nd.ndb_no where nd.src_cd = 10;	None	None	None	In the query plan the index scans weren't executed and it still ran in 0.034 ms therefore we did not think it was worth indexing	0	0	0	2991.55967	2991.55967	0	152.822587	152.822587	0	0.691	0.691	0	1447.071185	1447.071185	0	No changes were made	No changes were made	Almost every column in that query is a primary key in it's own table meaning that it is already automatically indexed and any additional indices may just distort these results.

18	Danya	select * from src_cd sc inner join nut_data nd on sc.src_cd = nd.src_cd inner join food_des fd on fd.ndb_no = nd.ndb_no where nd.src_cd in (1, 4, 5, 6, 7, 8);	nut_data(ndb_no)	Hash	CREATE INDEX idx_nut_data_ndb_no_hash ON nut_data USING HASH (ndb_no);	The query scans in a group of values that aren't a range for this column so we think a hash index can help quickly include and exclude values in this group	0	0	0	433.089638	347.835394	-85.254244	1267633.277	551651.6313	-715981.6458	1292.425	570.982	-721.443	0.773742	1.751374	126.351161	The Latency change was due to reasons that are not the index	Eventhough TPS increased by 126% , the index wasn't used meaning it probably had more to do with the laptop's available resources	This index was unsuccessful because it was not chosen. The original query without the search in certain values (Query 17) was already really fast and adding a hash index did not help the searching within groups, B+ Trees may have been a better choice for the clustering nature of this query.
19	Danya	select ndb_no, num_studies, nd.deriv_cd from deriv_cd dc inner join nut_data nd on dc. deriv_cd = nd.deriv_cd where dc.derivcd_desc like '% food%' order by nd.deriv_cd;	nut_data (deriv_cd)	B+ Tree	CREATE INDEX idx_nut_data_deriv_cd_btree ON nut_data (deriv_cd);	In the query we already have an index on the deriv_cd in deriv_cd but not in nut_data thererfore we chose to index it in hopes of speeding both the join and the order by clauses	40006	40006	0	303.353882	73.271363	-230.082519	130924.3489	574.962667	-130349.3863	133.147	80.381	-52.766	7.510699	12.44119	65.64623346	Relatively small latency difference, probably has more to do with the computer resources or internal planning than the actual query itself	Differences aren't huge which implies that the index wasn't fully utilized, however it may have made a small difference in the join, which could increase TPS	While the index was used, it may have not been the best or optimal choice for this query as it reduced the query execution time a lot more than it did planning time. This resulted in the different metrics being affected within a range that could also be put down to available resources (65% difference in TPS and 53 ms in latency)
20	Danya	select ndb_no, num_studies, nd.deriv_cd from deriv_cd dc inner join nut_data nd on dc. deriv_cd = nd.deriv_cd where dc.derivcd_desc like '% food%' and nd.deriv_cd = 'BFYN' order by nd.deriv_cd;	nut_data (deriv_cd)	B+ Tree	CREATE INDEX idx_nut_data_deriv_cd_btree ON nut_data (deriv_cd);	In the Query we have a join where dc.deriv_cd is indexed and nd.deriv_cd is not so by indexing nut_data(deriv_cd) the join, the pattern match and the order by should speed up	1	706	0	275.234842	225.620517	-49.614325	29891.46286	78490.27073	48598.80787	31.527	0.973	-30.554	31.722269	1029.698729	3145.980699	Eventhough execution time increased, overall latency decreased. This could have been to a combination of factors in the databse engine and the computers available resources	The index allowed for faster access to the rows where deriv_cd = 'BFYN', this reduced the number of tuples in the join causing overall execution to speed up and as a result more transactions per second	The index was successful, it was scanned and read many tuples, and TPS increased by a whopping 3145%. This suggests that it helped speed up the search by quickly filtering our rows where nd. deriv_cd did not match 'BFYN'.
21	Danya	select ndb_no, num_studies, nd.deriv_cd from deriv_cd dc inner join nut_data nd on dc. deriv_cd = nd.deriv_cd where dc.derivcd_desc like '% food%' and nd.deriv_cd = 'BFYN';	nut_data (deriv_cd)	B+ Tree	CREATE INDEX idx_nut_data_deriv_cd_btree ON nut_data (deriv_cd);	In the Query we have a join where dc.deriv_cd is indexed and nd.deriv_cd is not so by indexing nut_data(deriv_cd) the join and the pattern match should speed up	1001	746746	0	384.19588	76.465677	-307.730203	28433.78923	524.523048	-27909.26618	29.938	0.827	-29.111	33.409292	1213.577828	3532.455989	The query execution time decreased significantly, decreasing latency	The index allowed for faster access to the rows where deriv_cd = 'BFYN', this reduced the number of tuples in the join causing overall execution to speed up and as a result more transacctions per second	This index was successful for the same reason it was successful in Query 20. They both had to do very similar operations in terms of the columns being joined upon and the filtering conditions, so it makes sense to see similar results using the same index
22	Danya	select ndb_no, num_studies, nd.deriv_cd from deriv_cd dc inner join nut_data nd on dc. deriv_cd = nd.deriv_cd where dc.derivcd_desc like '% food%' and nd.deriv_cd = 'AI';	None	None	None	We didn't think it was worth indexing as it already operates at a really high efficiency	0	0	0	177.326716	177.326716	0	69009.02366	69009.02366	0	0.351	0.351	0	2859.274969	2859.274969	0	No changes were made	No changes were made	This query is almost identical to Queries 20 and 21 which means it could have also benefitted from the same B+ Tree index on nd. deriv_cd. However, when we found it already operating at 2860 TPS we thought that adding the index may not only be unnecessary, it may add overhead and slow down already really good perfomance.

Key	
Black Q Number	Was run 10000 times
Blue Q Number	Was run 1000 times

All time related values are measured in just difference in ms, TPS is measured as a percentage difference calculated as (after-before)/before *100 %