ADVANDB MCO1: Query Optimization

Ryan Austin Fernandez

CS-ST Student  
22A, E. Padua Street,  
BF Resort Village, Las Pinas City  
(+63)917-496-5071

ryan\_fernandez@dlsu.edu.ph

Marc Dominic San Pedro

CS-ST Student  
69A, Third Level  
Minas Tirith, Gondor  
(+63)977-817-5699

marc\_sanpedro@dlsu.edu.ph Jonah E. Syfu

CS-ST Student  
2419, Burgundy Transpacific Place,  
Taft Avenue, Fuyuki City  
(+63)915-620-3788

jonah\_syfu@dlsu.edu.ph

# ABSTRACT

Query optimization can have a very significant effect on the performance of a system, but this effect depends on a few factors.

The objective of the study is to determine what these factors are by constructing a few simple queries on the Community-Based Monitoring System or CBMS database, optimizing them using heuristic optimization, creating indices, views, and stored procedures, and testing the queries on different input values.

This is significant for future application of this knowledge can greatly improve the performance of the systems that are to be developed.

The scope of this study is optimization involving only the four methods mentioned.

The group was able to conclude the optimization is more effective when the query involved deals with more tables, each with select operations performed, on a database with appropriate indices available.

# INTRODUCTION

The group has built a simple application that interfaces with the CBMS database system of Palawan to extract certain data including the average number of overseas Filipino workers per nuclear family, the number of children above a given nutritional index in each geographical division, the average age of death per sex in each geographical location, the amount of fish of a particular type caught in each geographical division, crop densities, total aquatic animal catches per aquatic equipment used, and counts of common beneficiaries of particular classifications of Philhealth beneficiaries.

# ORIGINAL QUERIES

**2.1. Average OFW’s Per Nuclear Family**

This query is to determine the average number of OFW’s per nuclear family for each household that holds a nuclear family, divided by geographical location. The exact query is as follows.

SELECT mun, zone, brgy, purok, SUM(nnucfam) AS `Nuclear Families`, SUM(nofw) AS OFWs, SUM(nofw) / SUM(nnucfam) AS `Average OFW's per Nuclear Family`

FROM db\_hpq.hpq\_hh

WHERE nnucfam > 0

GROUP BY mun, zone, brgy, purok

HAVING SUM(nofw) > :count

The expected output is a series of geographical locations and their corresponding nuclear family counts, OFW counts, and the ratio between OFW’s and nuclear families. The query returned five hundred and fifty-seven rows and seven columns for :count = 0. Since the query does not have conditions that are too restricting, any general value given to :count resulted in the query running for an average of 129.945 seconds.

**2.2. Number of Children above a Particular Nutritional Index**

This query is to determine the count of children above a certain nutritional index in each geographical division. The query is as follows.

SELECT country\_resid, prov\_resid\_code, mnutind, COUNT(mnutind) nutCount

FROM hpq\_mem

WHERE mnutind <= :minimum\_index

GROUP BY country\_resid, prov\_resid\_code,mnutind

HAVING nutCount > :count

The expected output is a series of geographical locations and their corresponding number of kids above the given index. A <= operator is used since the hpq\_mem table uses 1 for the healthiest index and 4 for the lowest index. The query returned seventy-three rows and four columns for :minimum\_index = 4 and :count = 0. For the worst nutritional index, the query ran for 13.375 seconds. For the midlevel index, the query ran for 14.461 seconds. For the best nutritional index, the query ran for 12.359 seconds.

**2.3. Average Age of Death Divided by Sex and Geographical Location**

This query is to determine the average age of death per sex per geographical location. The exact query is as follows.

SELECT H.mun,H.zone,H.brgy, mdeadsx, AVG(mdeadage) avg\_death\_age

FROM hpq\_hh H, hpq\_death D

WHERE H.id = D.hpq\_hh\_id AND mdeady = :reason

GROUP BY H.mun,H.zone,H.brgy,mdeadsx

HAVING AVG(mdeadage) > :count

The expected output is a series of geographical locations, the two sexes, and their corresponding average death age. The query returned three hundred and eighty-six rows and five columns for :count = 0 and no specified :reason. For the most common cause of death “other”, the query ran for 5.625 seconds. For the median cause of death, “diabetes”, the query ran for 0.469 seconds. For the least common cause of death, “measles”, the query ran for 0.471 seconds.

**2.4. Amount of Fish per Type Caught**

This query is to determine the total count of fish caught in given geographical locations. The exact query is as follows.

SELECT H.mun,H.zone,H.brgy, SUM(aquani\_vol) fishcount

FROM hpq\_hh H, hpq\_aquani A

WHERE H.id = A.hpq\_hh\_id AND aquanitype = :type

GROUP BY H.mun,H.zone,H.brgy

HAVING COUNT(H.id) > :count

The expected output is a series of geographical locations and their corresponding counts for number of the given type of aquatic animal caught. The query returned three hundred and seventy-seven rows and four columns for :count = 0 and no :type specified. For the most common type of fish, “other”, the query ran for an average of 75.748 seconds. For the median type of fish, “milkfish”, the query ran for 1.922 seconds. For the least common type of fish, “tilapia”, the query ran for 0.859 seconds.

**2.5. Crop Densities**

This query is to determine the total crop volume, land area, and corresponding crop densities (volume/land area unit) per geographical division. The exact query is as follows.

SELECT H.mun,H.zone,H.brgy, SUM(crop\_vol) AS totalcrop, SUM(alp\_area) AS totalArea, SUM(crop\_vol)/SUM(alp\_area) AS cropDensity

FROM hpq\_hh H, hpq\_alp A, hpq\_crop C

WHERE H.id = A.hpq\_hh\_id AND H.id = C.hpq\_hh\_id AND croptype = :croptype

GROUP BY H.mun,H.zone,H.brgy

HAVING cropDensity > 0

The expected output is a series of geographical locations and their corresponding crop volume, land areas, and the ratio between crop volume and land area referred to as crop density. The query returned three hundred and ninety-four rows and six columns for :count = 0 and no specified :croptype. For the most common type of crop, “sugarcane”, the query ran for an average of 97.125 seconds. For the median type of crop, “corn”, the query ran for 55.406 seconds. For the least common type of crop, “palay”, the query ran for 14.714 seconds.

**2.6. Amount of Aquatic Animals Caught Per Type of Aquatic Equipment**

This query is to determine the number of aquatic animals caught per type of aquatic equipment used. The exact query is as follows.

SELECT mun, zone, brgy,SUM(aquaequip\_line) AS totalequip, SUM(aquani\_vol) AS totalvol, SUM(aquani\_vol) / SUM(aquaequip\_line) AS CatchPerEquip

FROM hpq\_aquaequip AA, hpq\_aquani AP, hpq\_hh H

WHERE aquaequiptype = :equip AND aquanitype = :animal AND H.id = AA.hpq\_hh\_id AND H.id = AP.hpq\_hh\_id

GROUP BY H.mun,H.zone,H.brgy

HAVING SUM(aquani\_vol) / SUM(aquaequip\_line) > :count

The expected output is a series of geographical locations and their corresponding counts of the specific type of aquatic equipment used, counts of aquatic animals caught, and the ratio of animals caught to number of equipment used. The query returned three hundred and seventy-five rows and six columns for :count = 0 and no specific type of animal or equipment specified. For the most common type of equipment and animal, “hooks and line” and “other”, the query ran for an average of 62.421 seconds. For the median type of equipment and animal, “gillnets” and “milkfish”, the query ran for 1.765 seconds. For the least common type of equipment and animal, “siftnet” and “tilapia”, the query ran for 0.766 seconds.

**2.7. Number of Common Philhealth Beneficiaries**

This query is to determine the counts per geographical location of citizens who are beneficiaries of Philhealth who are employed, individually paying, sponsored, and lifetime members. The exact query is as follows.

SELECT H.mun,H.zone,H.brgy,COUNT(H.id) benefCount

FROM hpq\_hh H, hpq\_phiheal\_spon\_mem PSM, hpq\_phiheal\_empl\_mem PEM, hpq\_phiheal\_indiv\_mem PIM, hpq\_phiheal\_life\_mem PLM

WHERE H.id = PSM.hpq\_hh\_id AND H.id = PEM.hpq\_hh\_id AND H.id = PIM.hpq\_hh\_id AND H.id = PLM.hpq\_hh\_id AND PSM.phiheal\_spon\_mem\_refno = PEM.phiheal\_empl\_mem\_refno AND PEM.phiheal\_empl\_mem\_refno = PIM.phiheal\_indiv\_mem\_refno AND PIM.phiheal\_indiv\_mem\_refno = PLM.phiheal\_life\_mem\_refno

GROUP BY H.mun,H.zone,H.brgy

HAVING benefCount > :count

The expected output is a series of geographical locations and their corresponding number of common beneficiaries in all the aforementioned Philhealth divisions. The query returned seven rows and four columns for :count = 0. Since the query does not have conditions that are too restricting, any general value given to :count resulted in the query running for an average of 0.126 seconds.

# QUERY OPTIMIZATION

## Heuristic Optimization

Four basic steps, according to Silberschatz (2010), were applied to the queries. The first was pushing all select operations lower into the tree. All tables in the FROM clause were then put into their own individual subquery and any WHERE clauses applied to the table were included in the subquery.

The second step was applying the most restrictive select operations first. The tables with WHERE clauses were the first to have their Cartesian products taken. If there was a tie, the priority was determined by the row count of the table. The table with less number of rows were joined first to decrease intermediate table sizes.

The third step was to transform Cartesian products into theta joins. This step was simple as it was just a syntactical change, eliminating the encompassing WHERE clause altogether.

The final step was to perform project operations early. In each of the base subqueries, the SELECT clause was changed from SELECT \* to only select the necessary columns, which were the columns that will be aggregated and the foreign key.

The performance of these five steps resulted in the heuristically optimized query.

**3.2. Indices**

The next approach was to add indices on the commonly referenced columns in the peripheral tables, specifically the columns used in the WHERE clause.

**3.3. Views**

Furthermore, certain operations on the relations were abstracted into views to allow for simpler queries. The base queries used were the heuristically optimized queries.

**3.4. Stored Procedures**

The final step was to encapsulate the entire query into a stored procedure, with the parameters abstracted as formal parameters to the procedure.

# RESULTS AND ANALYSIS

The general trend in the results was that heuristic optimization slightly improved the query’s performance; indices provided the biggest improvement; views and stored procedures provided marginal improvement.

Not all results were positive however. In the second section, query 1 and 7 are analyzed

**4.1. Positive Cases**

For queries 2 to 6, improvement was very explicit. Take for example, query 2: number of children above a particular nutritional index. The average performance, given the median nutritional index as a filter is as follows.

**Table 1 – Comparative Performance of Query 2**

|  |  |  |
| --- | --- | --- |
|  | **Average Runtime in Seconds** | **Improvement Percentage** |
| **Base** | 13.7093 | 0.0000 |
| **Heuristic** | 12.9437 | 5.5845 |
| **Indices** | 0.0077 | 99.9438 |
| **Views** | 0.0093 | 99.9322 |
| **Stored Procedures** | 0.0111 | 99.9190 |

As previously mentioned, heuristic optimization gave the query a slight 5.58% boost in performance speed, whereas adding indices on columns used in the WHERE clauses provided a 99.94% boost in performance speed from the original query. Views and Stored Procedures only provided a marginal decrease in performance.

Another exceptional case is query 4: amount of fish per type caught. The following table shows the result given the median input parameter of milkfish.

**Table 2 – Comparative Performance of Query 4**

|  |  |  |
| --- | --- | --- |
|  | **Average Runtime in Seconds** | **Improvement Percentage** |
| **Base** | 0.5342 | 0.0000 |
| **Heuristic** | 0.1516 | 71.6211 |
| **Indices** | 0.0125 | 97.6601 |
| **Views** | 0.0108 | 97.9783 |
| **Stored Procedures** | 0.0141 | 97.3605 |

As with query 2, heuristic optimization provided a boost, albeit much better than with query 2, while indices provided a large boost as well.

The similarities between queries 2 to 6 is that each of them has a select operation for each component table. For example, query 2 considers the nutritional index threshold to filter while query 4 considers the type of fish caught.

This select operation, having been performed before the joins, greatly lessened the number of rows involved in the theta join, greatly decreasing the size of the intermediate tables in the query.

For posterity, the statistics for the median parameter for queries 3, 5, and 6 follow.

**Table 3 – Comparative Performance of Query 3**

|  |  |  |
| --- | --- | --- |
|  | **Average Runtime in Seconds** | **Improvement Percentage** |
| **Base** | 5.9594 | 0.0000 |
| **Heuristic** | 5.6202 | 5.6918 |
| **Indices** | 0.0031 | 99.9480 |
| **Views** | 0.0048 | 99.9195 |
| **Stored Procedures** | 0.0093 | 99.8439 |

**Table 4 – Comparative Performance of Query 5**

|  |  |  |
| --- | --- | --- |
|  | **Average Runtime in Seconds** | **Improvement Percentage** |
| **Base** | 14.7141 | 0.0000 |
| **Heuristic** | 14.5332 | 1.2294 |
| **Indices** | 13.3780 | 9.0804 |
| **Views** | 13.3344 | 9.3767 |
| **Stored Procedures** | 13.5204 | 8.1126 |

**Table 5 – Comparative Performance of Query 6**

|  |  |  |
| --- | --- | --- |
|  | **Average Runtime in Seconds** | **Improvement Percentage** |
| **Base** | 0.1845 | 0.0000 |
| **Heuristic** | 0.0797 | 56.8022 |
| **Indices** | 0.0110 | 94.0379 |
| **Views** | 0.0109 | 94.0921 |
| **Stored Procedures** | 0.0078 | 95.7724 |

An explanation as to why query 4’s performance boost at the second step is the distribution of the aquanitype variable.

**Table 6 – Distribution of the aquanitype Variable**

|  |  |
| --- | --- |
| **aquanitype** | **Percentage of Rows** |
| 6 | 0.9747 |
| 3 | 0.0069 |
| 2 | 0.0051 |
| 4 | 0.0045 |
| 5 | 0.0044 |
| 1 | 0.0037 |
| NULL | 0.0007 |

There was more filtering done in the hpq\_aquani table before being joined with the hpq\_hh table, thus the size of the intermediate table was significantly less than in the base query.

Meanwhile, hpq\_mem(mnutind)’s distribution is quite even, with a lot of rows having null values, meaning the amount of rows being processed in the base query and the optimized query is approximately the same.

**Table 7 – Distribution of mnutind Variable**

|  |  |
| --- | --- |
| **mnutind** | **Percentage of Rows** |
| NULL | 0.8411 |
| 2 | 0.1510 |
| 3 | 0.0059 |
| 4 | 0.0011 |
| 1 | 0.0009 |

The hpq\_death(mdeady)’s distribution is also relatively even, as well as hpq\_crop(croptype)’s distribution.

**Table 8 – Distribution of the mdeady Variable**

|  |  |
| --- | --- |
| **mdeady** | **Percentage of Rows** |
| 17 | 0.3494 |
| 1 | 0.1656 |
| 5 | 0.0874 |
| 3 | 0.0649 |
| 8 | 0.0599 |
| NULL | 0.0474 |
| 11 | 0.0383 |
| 12 | 0.0333 |
| 10 | 0.0300 |
| 4 | 0.0300 |
| 9 | 0.0291 |
| 2 | 0.0241 |
| 6 | 0.0216 |
| 16 | 0.0092 |
| 13 | 0.0092 |
| 7 | 0.0008 |

**Table 9 – Distribution of the croptype Variable**

|  |  |
| --- | --- |
| **croptype** | **Percentage of Rows** |
| 1 | 0.5522 |
| 4 | 0.1923 |
| 3 | 0.1349 |
| NULL | 0.0914 |
| 2 | 0.0292 |

The reason why query 5’s performance is not as significantly improved as the others is because only one of its three component tables are being filtered. The ratio of the base query’s intermediate table sizes to the optimized query’s is still quite close to 1 since no filtering is being performed on the hpq\_alp table.

The reason why views and stored procedures caused a marginal decrease for most of the queries is because they do not exactly provide any physical improvement in the processing of the queries. They only provide a logical abstraction to the process, but the actual physical processing is categorically the same as with merely having indices in the tables. The performance might actually be worsened, since these logical abstractions require a quantifiable amount of overhead to execute.

**4.2. Negative Cases**

The following table shows the comparative results for query 1.

**Table 10 – Comparative Results of Query 1**

|  |  |  |
| --- | --- | --- |
|  | **Average Runtime in Seconds** | **Improvement Percentage** |
| **Base** | 129.9453 | 0.0000 |
| **Heuristic** | 133.9329 | -3.0687 |
| **Indices** | 129.8515 | 0.0722 |
| **Views** | 137.8624 | -6.0926 |
| **Stored Procedures** | 139.7031 | -7.5092 |

Indices provided a marginal improvement, but other methods worsened the performance of the query. The same can be said for query 7.

**Table 11 – Comparative Results of Query 7**

|  |  |  |
| --- | --- | --- |
|  | **Average Runtime in Seconds** | **Improvement Percentage** |
| **Base** | 0.1263 | 0.0000 |
| **Heuristic** | 0.1341 | -6.1758 |
| **Indices** | 0.1422 | -12.5891 |
| **Views** | 0.1656 | -31.1164 |
| **Stored Procedures** | 0.1172 | 7.2051 |

To better analyze the reason for this performance, here is the optimized query 1.

SELECT mun, zone, brgy, purok, SUM(nnucfam) AS `Nuclear Families`, SUM(nofw) AS OFWs, SUM(nofw) / SUM(nnucfam) AS `Average OFW's per Nuclear Family`

FROM (SELECT mun, zone, brgy, purok, nnucfam, nofw

FROM db\_hpq.hpq\_hh

WHERE nnucfam > 0) A

GROUP BY mun, zone, brgy, purok

HAVING SUM(nofw) > :count

In query 1’s case, the only difference from the original query in Section 2.1 is the subquery that performs a projection after a selection operation. There is no categorical difference in the physical processing of the query. In fact, having an additional step in the form of the subquery actually worsened the performance of the query. With regards to the intermediate tables, the size was still relatively the same as with the base query since the WHERE clause has a static condition instead of a dynamic one and the HAVING clause is only processed after the aggregation.

Similarly, here is optimized query 7.

SELECT H.mun,H.zone,H.brgy,COUNT(H.id) benefCount

FROM (SELECT id, mun, zone, brgy

FROM hpq\_hh) H INNER JOIN

((((SELECT hpq\_hh\_id, phiheal\_spon\_mem\_refno

FROM hpq\_phiheal\_spon\_mem)PSM

INNER JOIN

(SELECT hpq\_hh\_id, phiheal\_empl\_mem\_refno

FROM hpq\_phiheal\_empl\_mem )PEM

ON PSM.hpq\_hh\_id = PEM.hpq\_hh\_id

AND PSM.phiheal\_spon\_mem\_refno = PEM.phiheal\_empl\_mem\_refno)

INNER JOIN

(SELECT hpq\_hh\_id, phiheal\_indiv\_mem\_refno

FROM hpq\_phiheal\_indiv\_mem )PIM

ON PEM.hpq\_hh\_id = PIM.hpq\_hh\_id

AND PEM.phiheal\_empl\_mem\_refno = PIM.phiheal\_indiv\_mem\_refno) INNER JOIN

(SELECT hpq\_hh\_id, phiheal\_life\_mem\_refno

FROM hpq\_phiheal\_life\_mem ) PLM

ON PIM.hpq\_hh\_id = PLM.hpq\_hh\_id

AND PIM.phiheal\_indiv\_mem\_refno = PLM.phiheal\_life\_mem\_refno )

ON id = PSM.hpq\_hh\_id

GROUP BY H.mun,H.zone,H.brgy

HAVING benefCount > :count

Aside from converting Cartesian products to theta joins, the size of the intermediate tables are the same due to there being no filtering being done in the peripheral tables. The exact same rows are being used in the base query as in the optimized query.

The fact that physically, the only difference in the processing is the overhead of additional subqueries and joins, which outweighs the theoretical improvement of the query, is the reason as to why the optimized query performs worse than the base query.

In addition to this, indices, which are not being checked as much but have to be consulted, only add overhead to the query, in query 7’s case, since an index was added for each of the four Philhealth tables used, whereas query 1 only had an index on hpq\_hh(nnucfam). The physical implementation of views also worsened the performance in both queries.

# CONCLUSION

Query optimization can greatly improve the performance of an application for up to 99.94% in terms of speed. However, it greatly depends on the structure of the queries, the structure of the database, and the number of tables in each query.

The structure of the queries affects the level of optimization based on the amount of select operations performed on each table. The more selection operations, the better the increase in performance due to intermediate tables having much less rows as opposed to the original query.

The structure of the database can also affect the performance since a normalized database would generally have less data to process at any time since columns have been delegated to other tables. Additionally, the presence of indices on appropriate rows i.e. rows used in WHERE clauses can greatly improve the performance of a query.

Finally, the number of tables can affect the optimization. Single table queries, when optimized, only lessen one dimension of the intermediate tables, while queries with two or more tables with select operations, once pushed down the process tree, can lessen the intermediate table sizes in two or more dimensions, greatly lessening the overhead in processing the query and improving its performance.

In conclusion, optimization is best performed in operations dealing with numerous filters and tables on a database that is properly normalized and has the proper indices.

# REFERENCES

1. Silberschatz, A., Korth, H. & Sudarshan, S. (2010). Database System Concepts, 6th Edition, McGraw-Hill Book Co. Ding, W. and Marchionini, G. 1997. *A Study on Video Browsing Strategies*. Technical Report. University of Maryland at College Park.