Using the Execution Plan/Explain Plan to Enhance Custom Field Performance in IFS10 at Teknos Group OY

Renz Razon

2021 Laurea

Laurea University of Applied Sciences

Using the Execution Plan/Explain Plan to Enhance Custom Field Performance in IFS10 at Teknos Group OY

Renz Razon
Degree Program in Business
Information technology
Bachelor's Thesis
November, 2021

Laurea University of Applied Sciences
Business Information Technology

Renz Razon

Abstract

Using the Execution Plan/Explain Plan to Enhance Custom Field Performance in IFS10 at Teknos Group OY

Year 2021 Pages 40

It is vital for the performance of an ERP system to have optimally performing SQL queries. Teknos Group Oy is in the process of upgrading the company's IFS ERP solution during which Custom Fields in the form of SQL queries are implemented to extend the base functionality of the ERP system to meet user and customer needs. The purpose of this thesis project was to enhance the performance of SQL Custom Fields using the execution plan. The objective was to create an SQL test case to demonstrate the use of an execution plan in identifying suboptimal procedures.

The tools and methodology used for this research revolved around the need for improving the performance of an SQL query. They consisted of an execution plan, PL/SQL Developer Tool, and analysis. The execution plan, which is the blueprint of an SQL's operation, is presented to gain a base understanding of how the operation is created and the decision-making process behind its creation. The PL/SQL Developer Tool was used to acquire the execution plan and navigate through its procedures step-by-step. Root Cause Analysis was the methodology used to identify performance issues in the execution plan procedures and to use the results to improve and refactor the SQL code.

The results show that the execution plan is a useful tool in identifying underlying SQL performance issues and areas of improvement. Operations such as Full Table Scans, Cartesian Joins, High cost due to a select all statement, and Index Range Scans were all identified. These results were taken into consideration during the SQL code refactoring process, resulting in an overall better execution plan with lower cost and efficient index access methods. Knowing how to interpret its results and why such operations are chosen by the optimizer is key to identifying the correct issues.

Keywords: ERP, Custom Fields, SQL, Execution Plan, Optimizer

Table of Contents

1 Introduction

Enterprise Resource Planning (ERP) systems are integral in streamlining and unifying an organizations data into one central database. These types of systems allow business processes to fluidly interact and exchange data with one another creating a more agile business that can adapt better to changes. This becomes increasingly important as a business grows to meet more demand. Handling redundant and time intensive tasks are automated, reducing the amount of workload of employees allowing them to focus on other important matters. With that, the importance of ensuring that the creation of these data do not hamper the workflow of its users.

Some ERP systems allow businesses to tailor their needs by allowing its base functionality to be extended through customization features. These types of customizations can be in different forms either to improve the user interface and user experience or to create custom data specific to the company or customer's needs. This thesis describes the use of the Execution Plan and in what ways it can be utilized to enhance an SQL Custom Objects performance for the ERP system of the client company. The Execution Plan is described in detail of its functions and usage using the client company's tool. These tools were then used to troubleshoot and improve an SQL query's performance.

1.1 Company Background

The client for this thesis project is Teknos Group Oy. Teknos Group Oy is a family-owned global coatings company founded in 1948 in Tuomarila Espoo, Finland. The company offers a wide variety of paint and coating solutions from Industrial, professional, and local consumers (Teknos history, n.d.). With the slogan "Making the world last longer", the company prides itself on advance and sustainable paint and coating solutions to protect and prolong all sorts of products and surfaces from the simplest to the harshest of elements possible. The company employs over 1700 employees as of 2018 in over 20 countries spanning from North America, Europe, and Asia. The company produces over 100,000 tons of paint per year and a net sale of 398 million euros as of 2019 (Teknos key figures, n.d.).

1.2 IFS ERP System

An Enterprise Resource Planning (ERP) system is an integration of all business processes into one software. It is designed to contain the business processes of a company such as Sales, Supply Chain, Human Resources, Accounting and many more. Each business process has its own activities that take in some input and produce an output such as reports or useful data that will be stored in a common database shared to other business processes and even external tools. ERP solutions tend to offer a variety of automation features to ensure that the users

can focus on other important tasks. This integration of business processes streamlines the flow of data from one department to another, ensuring that the business functions smoothly.

Industrial and Financial Systems (IFS) is a developer and provider of ERP solutions. IFS ERP solution specializes in manufacturing and distribution of goods, asset maintenance, and manage service-focused operations (Get to know IFS, n.d.).

One of the strong features that IFS provides its customers is the ability to add their own customizations through Custom Objects. Custom Objects makes it possible to extend the base functionalities and features using Custom Fields, Context Menu Items, Pages, Tabs, and more (About Custom Objects, n.d.).

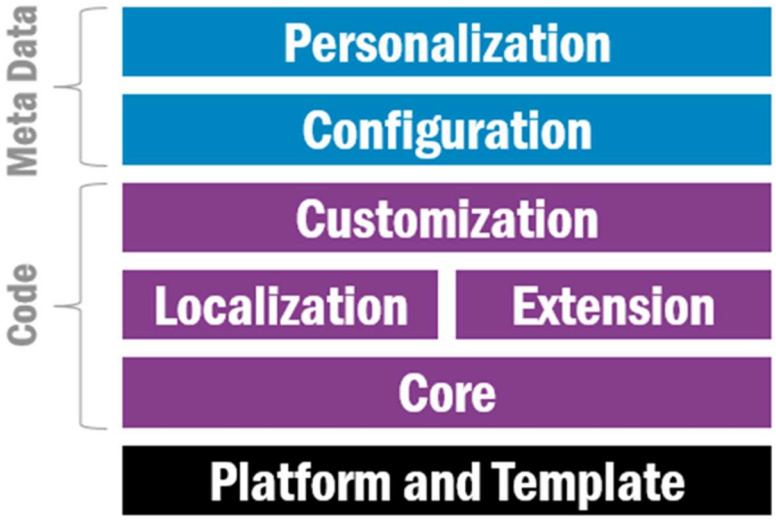


Figure 1: IFS Layered Application Architecture (Layered Application Architecture, n.d.)

Custom Objects are done in the ERP software's Configuration Layer of its 6 Layered Application Architecture (LAA). This architectural design tackles the problem of separation of ownership and makes extending features for the customers unintrusive to the base code (Layered Application Architecture, n.d.).

ERP systems like IFS are complex pieces of software that offer numerous functionalities that would meet small, medium, to large scale business requirements. It is a centralized system where many business transactions are processed and stored in a central database for every business function to interact with, streamlining the work process. Having the ability to customize the solution to further fit the business needs is an important feature to ensure that the business will not always try to fit the software.

1.3 Project Background

In 2018, Teknos Groups ICT Team initiated a project of upgrading the company's IFS ERP solution from IFS version 7.5 to IFS version 10. This upgrade project was planned to be deployed to all countries that Teknos is situated on in a series of country roll out projects. With the arrival of IFSv10, comes its features of creating in house solutions to meet company needs using Custom Objects. Creating custom objects in 7.5 was very limited and new implementations such as Custom Fields would always go to IFS for development. Having the feature as part of the IFSv10 allowed Teknos to create Custom Object solutions such as Custom Fields, Menus, Information Cards, Logical Units, Pages, Enumerations, and Tabs. Custom Objects such as Custom Fields and Custom Menus are written using PL/SQL or just SQL, while others are user interface based.

1.4 Problem Statement

With the ability to create the solutions in-house, comes the problem of ensuring that the solutions are meeting certain performance standards, more specifically for SQL queries Custom Fields. Due to the many SQL customizations being implemented, performance issues have been reported to occur on certain processes in the system. Investigations by the internal and external development team have concluded that some of these performance related issues are caused by some SQL custom fields, as well lack of indexes on the tables being used. With that, the clients want to find out why the SQL customization are causing such issues and how can the SQL performance be improved.

1.5 Objectives

Key Objectives

- To utilize the Execution Plan to enhance an SQL Customs Fields query's performance.
- To create an SQL test case and identify suboptimal execution plan procedures.

1.6 Scope and Limitation

The scope of this thesis will focus on understanding, interpreting, and improving the estimated execution plan results such as identifying missing indexes, improper join operations, and expensive operations to tune an SQL query's performance using the PL/SQL Developer tools Explain Plan feature. The actual runtime performance will not be tested and will just focus on a before and after result of the Execution Plan after the SQL code has been modified. Other tuning methods such as Tracing, Server Operating System Tuning will not be tackled.

2 Methodology and Tools

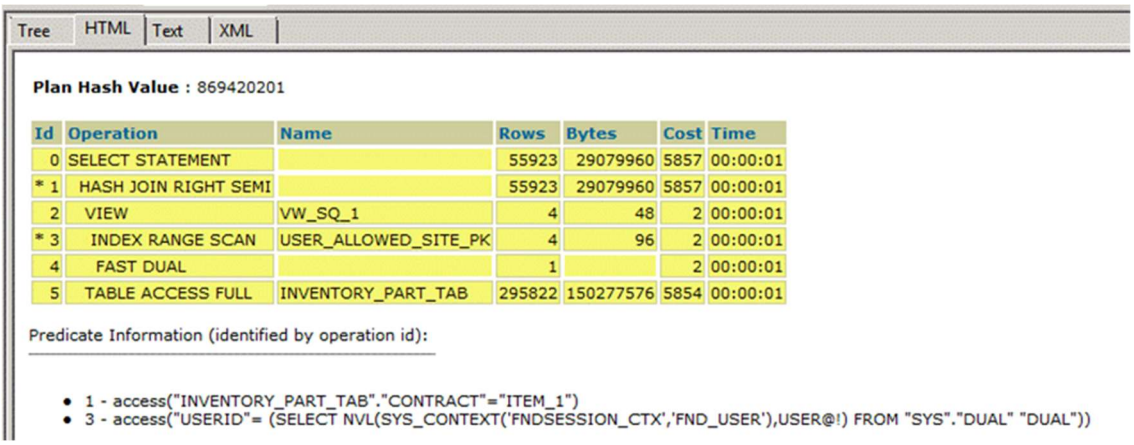
This chapter introduces the methods and tools to be used to tune SQL queries. The Execution Plan, what is the Execution Plan, the optimizer, and its components, how the optimizer chooses the execution plan and how it decides to create the plan, what are the factors it considers in creating an execution plan. The PL/SQL Developer Tool, how to use the tool to view the plan and indexes, how to read the Execution Plan, as well as interpreting the plan results. Lastly, the Root Cause Analysis, the methodology used to investigate and tune the SQL using the Execution Plan.

2.1 Execution Plan/Explain Plan

Unlike programming software code where a developer has control on what is written and how code is executed. SQL code, even with its declarative nature, its execution is not always decided by the developer. To understand how a SQL is ran behind the scenes and it process flow, one must look at the Execution Plan. The Execution Plan is the blueprint on how SQL is processed step-by-step and is a very useful tool to identify these processes behind the scenes and trouble shoot the SQL to improve the plan.

2.1.1 Execution Plan

According to the Oracle Tuning Documentation, the execution plan is a sequence of steps that the database would perform to run a SQL statement in fetching data from the database tables (Oracle, 2021, p.122). Knowing these steps, we can have a better understanding on how the SQL is executed and key metrics of the resources it is using during the process. The key metrics include the system resources (I/O, CPU, RAM), number of rows an operation is returning (Cardinality), what type of access method it is using, logical sequence, access filters, and more. Understanding these metrics are key to the SQL tuning process.


Tree | HTML | Text | XML

Plan Hash Value : 869420201

Id	Operation	Name	Rows	Bytes	Cost	Time
0	SELECT STATEMENT		55923	29079960	5857	00:00:01
* 1	HASH JOIN RIGHT SEMI		55923	29079960	5857	00:00:01
2	VIEW	VW_SQ_1	4	48	2	00:00:01
* 3	INDEX RANGE SCAN	USER_ALLOWED_SITE_PK	4	96	2	00:00:01
4	FAST DUAL		1		2	00:00:01
5	TABLE ACCESS FULL	INVENTORY_PART_TAB	295822	150277576	5854	00:00:01

Predicate Information (identified by operation id):

- 1 - access("INVENTORY_PART_TAB"."CONTRACT"="ITEM_1")
- 3 - access("USERID"= (SELECT NVL(SYS_CONTEXT('FNDSESSION_CTX','FND_USER'),USER@!) FROM "SYS"."DUAL" "DUAL"))

Figure 2: Execution Plan of a SQL query (SELECT * FROM ifsapp.inventory_part)

The image above shows an execution plan for an SQL query SELECT * FROM ifsapp.inventory_part. The query to fetch all data in the inventory part table. Each row on the plan corresponds to an operation that will be executed by the optimizer to get the requested results, which in the example, fetches all the data in on the inventory part table.

The Execution Plan default columns include the following ID, Operation, Name, Rows, Bytes, Cost, and Time. The columns can be changed to add more options.

ID –The ID is simply an identification for the operation. It does not correspond to the sequence on how the plan is going to be read.

Operation – Is the task that will be executed by the optimizer. This can be a table access or join operation.

Name – Contains the name of a table, index table, view being accessed during the operation.

Rows – Estimated number of rows returned on the operation or Cardinality.

Bytes – Corresponds to the estimated data of the entire operation and is determined by the amount of data being fetched in Bytes. This is affected by the number of columns selected and rows fetched.

Cost – Cost is a value assigned by the Optimizer. This value is determined by the Optimizer based on several factors such as Cardinality, processing, type of access method and more. This value is important to the Optimizer to decide what plan would suite best in executing an SQL query.

Time – Time is the estimated time the operation will run.

What is happening on the execution plan above is that on ID 5, the plan shows the operation TABLE ACCESS FULL, which simply fetches all data as the query requested with no WHERE clause, on the table INVENTORY_PART_TAB where the data is stored. ID 3 is more of a permission check; it shows an INDEX RANGE SCAN on USER_ALLOWED_SITE_PK where it returns the RowID's of sites the user is allowed access to and filters out restricted sites. If we look at our query compared to the table being accessed in ID5, we are fetching from the IFSAPP.INVENTORY_PART, while the execution plan shows INVENTORY_PART_TAB, this is because of IFS's design to hide certain implementations of the system. Also, we can see an inside look to the filter system that IFS has on ID 3.

By looking at the execution plan, we can begin to understand how the SQL statement is processed step-by-step behind the scenes and answer questions such as how much resource it might use, is it returning the correct number of rows, how is it accessing tables (Is the SQL

taking advantage of an Index) and understand how it is logically executed. The results in turn, can be leveraged for troubleshooting performance issues or actively look for ways to re-write and improve a queries performance. In the next subchapter we will look at the optimizer and how it creates an execution plan.

2.1.2 What is the Optimizer

The optimizer is an integrated tool in the database software, and it determines the most optimal method of executing an SQL statement in retrieving and accessing the data being requested (Oracle, 2021, p.55). It is the driving force in creating, choosing, and comparing execution plans that would be the most appropriate based on the cost of the entire each plan it creates (Oracle 2021, p.55). The plan determines this cost through, Cost Based Optimization (CBO). Before Oracle 10g, the optimizer had two ways of creating an execution plan, Rule Based Optimization (RBO) and Cost Based Optimization (CBO). RBO is a deprecated approach and from Oracle 10g onwards and for this thesis we will be only focusing on CBO.

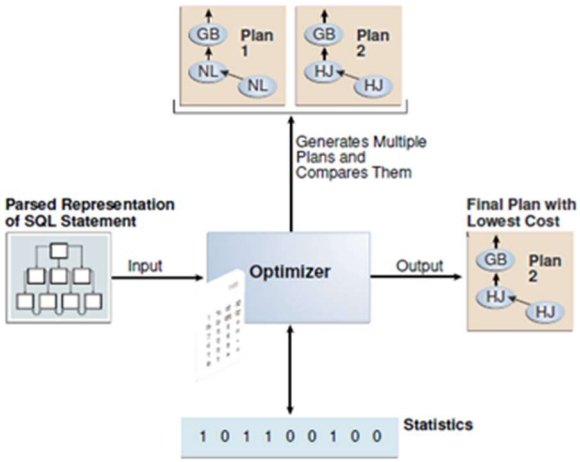


Figure 3: Process of creating an execution plan (Oracle, 2021, p.56)

2.1.3 Cost Based Optimization (CBO)

In determining which plan would suite best for the SQL query, the optimizer uses a cost-based approach when comparing the Execution Plans it created. CBO assigns a cost to each of the number of factors and assigns an overall cost which is the estimated resources that a query will use when making the final decision (Oracle 2021, p62-p63). Factors the Oracle optimizer considers in determining the overall resource cost of an SQL query are as follow:

1. System resources (I/O, CPU, Memory)

2. Number of rows expected to be returned (Cardinality)

3. The size of the data set (Number of rows the SQL is working with)

4. Data distribution

5. Access structures (Indexes)

CBO relies on data statistics to create an optimal execution plan. Ensuring that the factors are kept up to date is crucial in ensuring that the optimizer can create the lowest cost execution plan using the most efficient access paths and join methods and that the overall database performance can be kept at an optimally good level. System hardware are powerful enough to handle the given loads, the database statistics are kept up to date and the optimizer uses the latest stats for the execution plans.

2.1.4 Optimizer Components

The optimizer is divided into three components. The Query Transformer, Estimator, and Plan Generator. Once a query has been successfully parsed it will then be handed to the optimizer to create the execution plan. Figure 4 shows the three steps and decision-making process of the optimizer in creating and comparing execution plans.

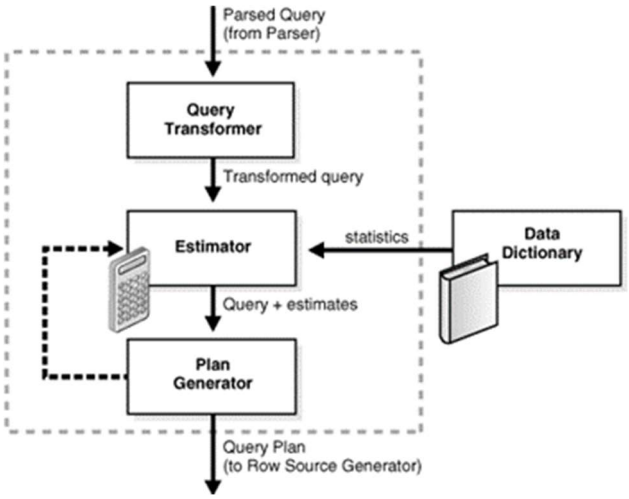


Figure 4: Oracle Optimizer Components (Oracle, 2021, p.58)

2.1.4.1 Query Transformer

In this step, the optimizer checks for usable and semantically the same alternatives of the SQL and decide whether it will yield lower cost. Query transformations look for possibly more efficient access path (Oracle, 2021, p.58).

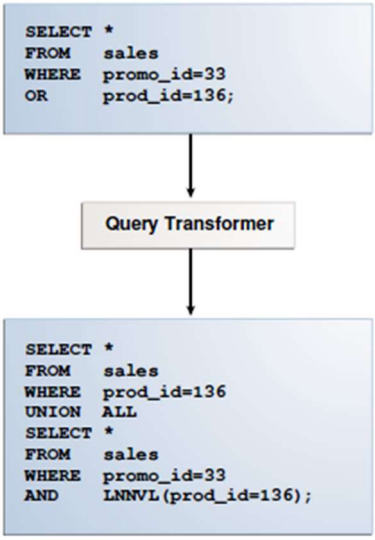


Figure 5: Optimizer performing an OR Expansion to a SQL statement (Oracle, 2021, p.59)

The query transformation above shows an OR Expansion, one of the 9 query transformations methods used by the oracle optimizer. This method is used when the optimizer finds a WHERE clause containing an OR operator and transforms it into a UNION ALL (Query Transformations, 2013). The transformed query will be processed separately to check whether its cost is lower compared to the original.

2.1.4.2 Estimator

After the Query Transform process, whether the statement has been transformed. The estimator component will calculate the total cost of the execution plan. The estimator relies on table statistics for this step and will determine the overall cost using the three measures (Oracle, 2021, p.59):

1. Selectivity - Selectivity is calculated as follow (number of rows selected/total rows). It is the fraction of rows returned from a row set. It is a value between 0 and 1, where 1 represents low selectivity meaning that 100%(all) of rows are selected from an operation, a value below 1 is headed towards higher selectivity (Gogia, 2017).

2. Cardinality - Cardinality is the selectivity * total number of rows. It is the total number of rows returned (Gogia, 2017).

3. Cost - Cost is the overall system resources that a query is predicted to use such as I/O, CPU, RAM, including the Cardinality estimates, and access structures (Oracle, 2021, p.59-60).

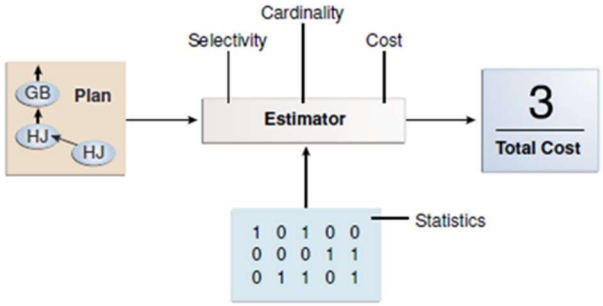


Figure 6: How the estimator calculates the cost of a plan (Oracle, 2021 p.60)

The factors mentioned above play an important role in ensuring that an optimal execution plan will be created. In particular the cardinality estimates, oracle emphasizes should be as accurate as possible because of its influence in the overall execution plan. The optimizer relies on accurate and up to date statistics to determine the cost of a join, cost of sorting, and access method it will use. If the statistics are out of date, the optimizer might decide to use other access methods and joins that can cause suboptimal performance.

2.1.4.3 Plan Generator

To find the most efficient access methods and join types to be used, the plan generator will try different access paths, join methods, and join orders for each query block using various combinations (Oracle, 2021, p.63-65). The method with the lowest cost in this step will be chosen by the optimizer.

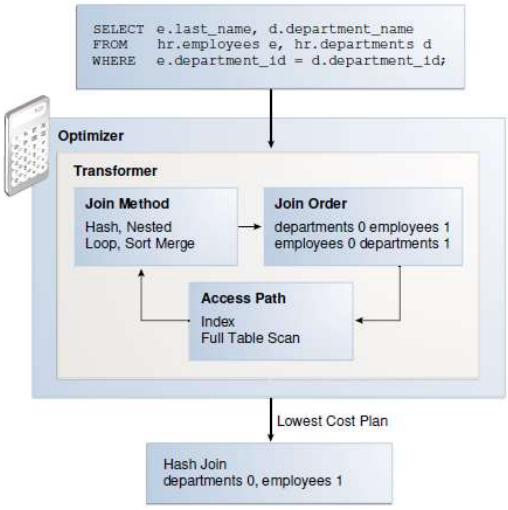


Figure 7: Optimizer Plan generator trying different access paths and returns the lowest cost (Oracle, 2021, p.64)

2.2 Root Cause Analysis (RCA)

The methodology chosen for the SQL tuning process of this thesis is the Root Cause Analysis (RCA). The purpose of RCA is to identify the root cause of a problem that can be corrected and when corrected will be able to prevent the fault from reoccurring in the future. RCA is not a clearly defined methodology, but rather a process for problem-solving that can be molded to fit the needs of the users for issues such as investigating an identified incident, problem, concern, or non-conformity (Root Cause Analysis: Process, Techniques, and Best Practices, 2021). The point clearly shows due to the varying of steps and processes when searching RCA methods online. Depending on the requirements and the subject, the steps to perform RCA will range from a 4 step process up to a 10-step process (See Figure 8). Ultimately though, these models would still overlap ideas and steps in performing RCA such as (1) Problem identification, (2) Diagnosing the problem, (3) Implementation a solution, and (4) Maintaining results (Okes, 2009). Having a well-structured meth-od is important to further maximize the benefits per-forming RCA.

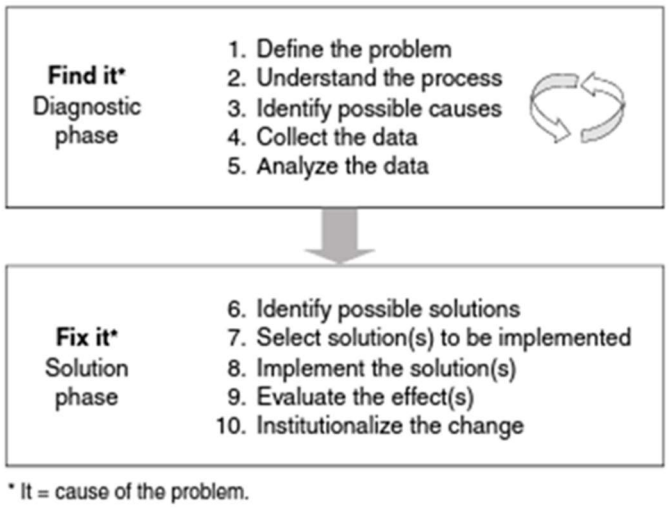


Figure 8: Duke Okes 10 step RCA model - DO IT2 Problem Solving Model (Okes, 2009)

To further understand what RCA is, we need to define what a Root Cause is. In James' and Lee's paper Root Cause Analysis for Beginners, they defined Root Cause in four different ways (Rooney and Vanden Heuvel, 2004):

1. Root causes are specific underlying causes.

2. Root causes are those that can reasonably be identified.

3. Root causes are those management has control to fix.

4. Root causes are those for which effective recommendations for preventing recurrences can be generated.

In other words, the root cause is the specific/origin cause of a problem. This is different from the general cause such as for example human error, or equipment failure. The actual root cause should be identifiable, controllable, and can be prevented from reoccurring when a solution has been implemented. The Analysis in RCA is the problem-solving process.

2.2.1 Chosen RCA Method

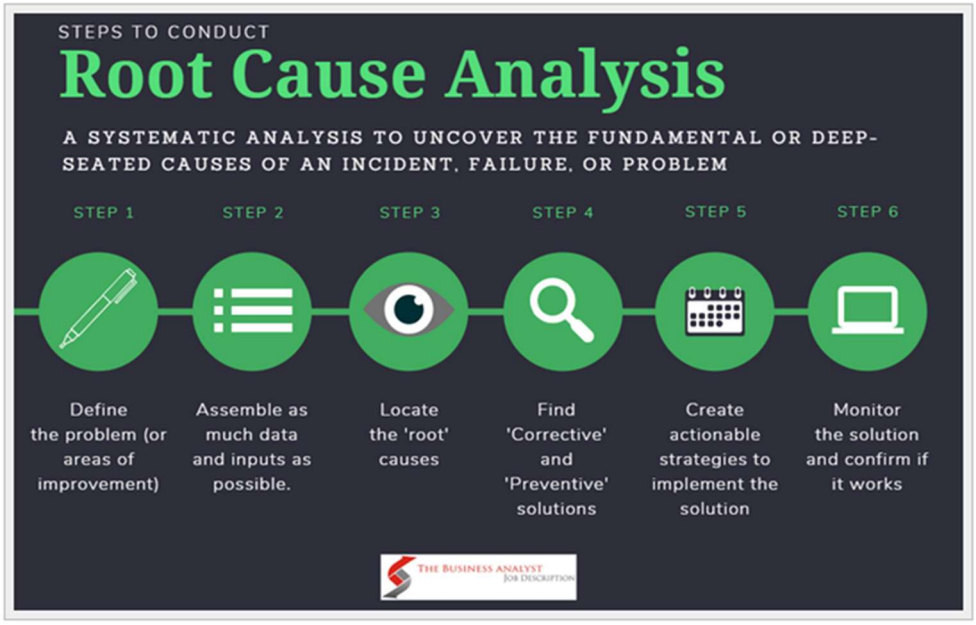


Figure 9: The Business Analyst 6 Step RCA model (Root Cause Analysis: Process, Techniques, and Best Practices, 2021)

The chosen RCA method for this research will be a 6-step process introduced by the The Business Analyst Job Description website. The rea-son for this is due to the model's simplicity for a problem-solving process and sequential approach in tuning SQL using the Execution Plan as well the similarities that the model has with Oracles SQL tuning Steps.

Oracles SQL Tuning model would include the following in tuning process:

1. Identifying high-load SQL statements
2. Gathering performance-related data
3. Determining the causes of the problem
4. Defining the scope of the problem

5. Implementing corrective actions for Sub optimally performing SQL statements
6. Preventing SQL performance Regression (Oracle, 2021, p29-30).

The two approaches have very common procedures. Compared to the business analysts RCA model, Oracle's process differs on step 4 which is defining the scope. For this research, the scope is already defined and is focused mainly on SQL changes that can improve an SQL's performance based on the provided Execution Plan.

2.3 PL/SQL Developer Tool

This sub-chapter introduces the PL/SQL Developer Tool. It is the IDE (Integrated Development Environment) that the Teknos development team uses to create SQL Custom Objects for the ERP System. It features an easy and interactive way to view the Execution Plan without having to run the EXPLAIN PLAN command.

2.3.1 How to view the Execution Plan

The PL/SQL Developer Tool is an IDE created by Allround Automations, a company whose aim it to provide feature rich tools for Oracle developers. The IDE features a wide range of tools for developing, testing, debugging, and optimizing Oracle PL/SQL store program units (packages, triggers, etc) (About us - Allround Automations, 2021). It is the IDE that the Teknos internal ERP development team uses for writing SQL code used for the ERP customizations. It also features an easy way of viewing the execution plan which is most suitable for this thesis.

Allround Automations PL/SQL Developer Tools provides an easy and interactive way of viewing an execution plan. One can view an execution plan by running PL/SQL code in the Explain Plan window by going to File tab, select New, and chose the Explain Plan window. This will open a window where you can enter your PL/SQL or SQL code and hit Execute on the Session Tab to view the query's execution plan.

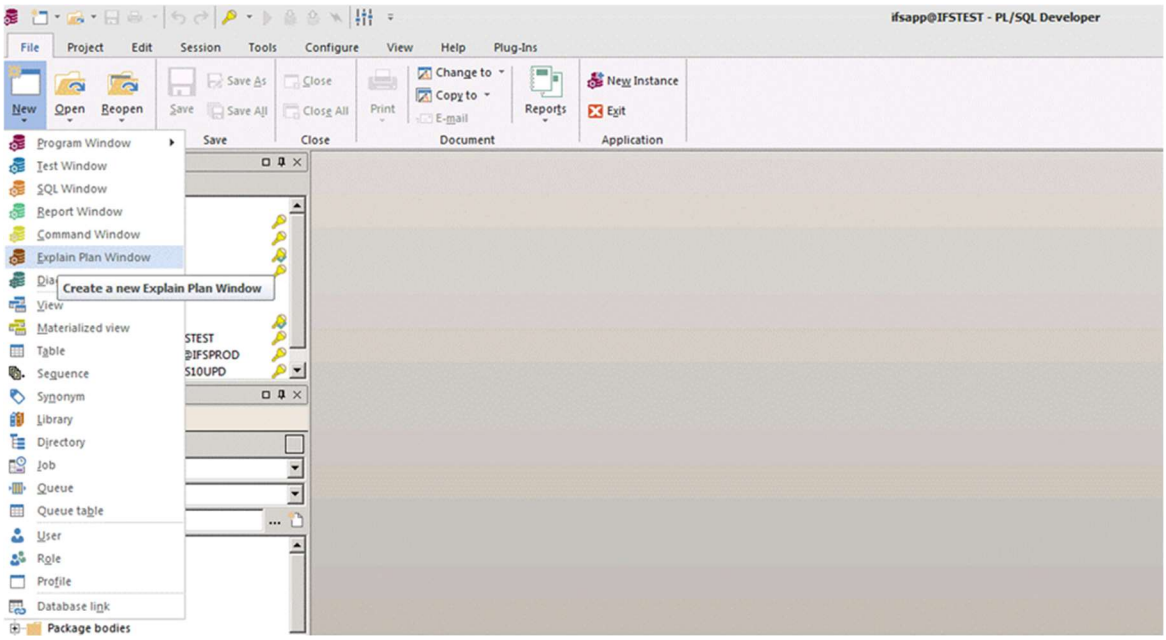


Figure 10: Creating an Execution Plan Window in PL/SQL Developer Tool

2.3.2 Viewing the Execution Plan using the Explain Plan Window

For the case of the client, to view the execution plan requires the use of the “ifsapp” user account connected to an environment. The “ifsapp” user account has the privileges to view execution plans and access restricted tables. The Teknos development teams account which has all the high permissions and access is not given access to these features.

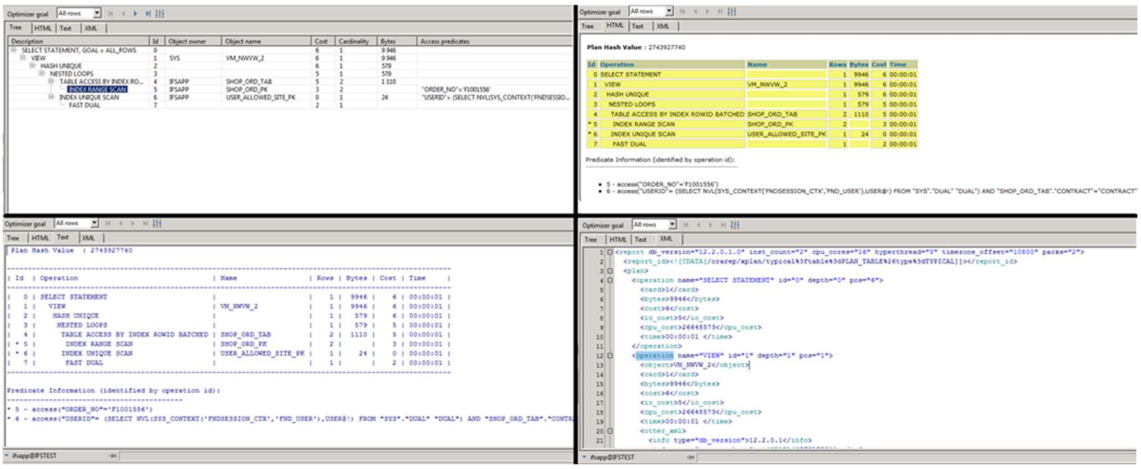


Figure 11: Execution Plan of Running "SELECT * FROM ifsapp.shop_ord WHERE order_no = 'F1001556'"

Running the SQL in Figure 11 on the Explain Plan Window will result in 4 different views of the execution plan, Tree, HTML, Text, and XML. The Tree view offers a rich feature of easily adding more variables in the preferences setting as well as the ability to traverse the plan step-

by-step using the navigation button on the windows header. HTML and TEXT will only display columns, ID, Operation, Name, Rows, Bytes, Cost, Time and not include the plan navigation buttons. The XML version can be exported to be used on other programs that can convert the XML to a plan. The user can also set optimizer goals to be Cost Based (First rows, All Rows) or Rule Based (Rule), by default this is set to All rows.

2.3.3 Viewing the available indexes

Similar to an Index in a book where you can find sorted keywords together with the pages they are located on, indexing makes searching information much faster without needing to flip the pages of the book one by one. Indexes in a table function the same way and makes retrieving data much faster because the data is sorted and contains the pointer to where the data is in memory. Without an index, queries must read the entire table to look for information (Oracle Indexes and types of indexes in oracle with example - Techgoeasy, 2019). There are different types of indexes, and this thesis will not be covering Indexes in-depth.

Viewing the index can be done in two ways. First, is from the Tree view of the plan window, by Right Mouse Button (RMB) the Object Name and Selecting View, would return the table information such as indexes available to use (See Figure 12). The second way is by typing the object name on a SQL or Explain Plan window, Highlight the name of the Object, RMB, then Select View.

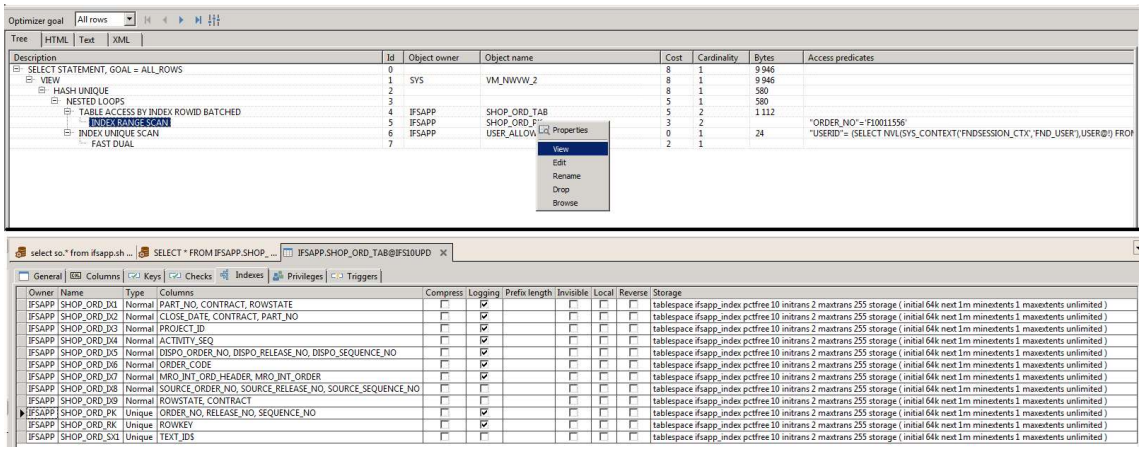


Figure 12: Viewing the available indexes of an Object from the Execution Plan

Identifying which indexes are available is important to utilize the fastest way searching through the tables. Which is important to making sure that SQL runs fast and that the optimizer chooses the best access path on the plan. In figure 12, SHOP_ORD_PK shows that the index is Type unique, meaning that when we supply all the 3 columns mentioned (ORDER_NO, RELEASE_NO, SEQUENCE_NO) on the Columns column on the WHERE clause, would results on

querying a unique row. By using the other indexes, would results in an index range scan because there can be multiple rows returned.

2.3.4 Reading the Execution Plan

Although the PL/SQL Developer Tool can help us traverse the Execution Plan step-by-step. It is still important to know how to read it without relying on the tool.

The Execution Plan is a tree like structure where the data flows upward from the leaves to the Root or from Child Node to the Parent Node. This relationship can be seen from the indentation of the Operation Column. The database uses a depth-first search approach where it starts from the top of the plan (Root: SELECT STATEMENT) and working its way down the tree until it ends up on a leaf with no branches. The operation for that leaf is then executed, then it goes back up the tree, performing the operations along the way, then traverses back down to unvisited leaves. This process is repeated, walking down the tree until all leaves are visited, walking back up to a different branch until and all steps are read (Saxon, 2020).

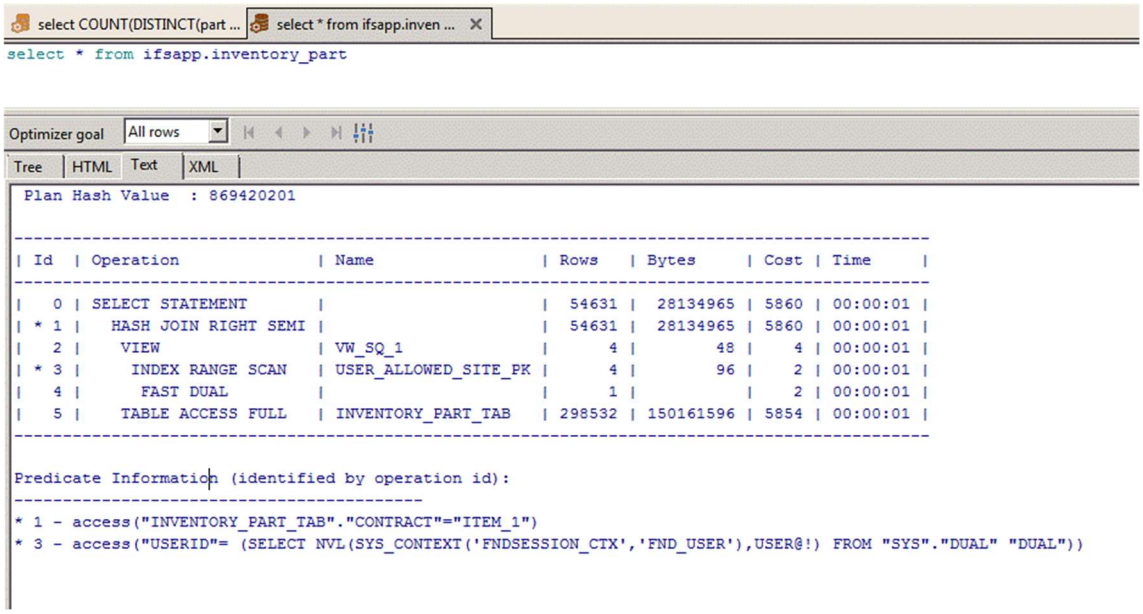


Figure 13: Execution Plan for (SELECT * FROM IFSAPP.INVENTORY_PART)

The execution plan data flow is as follow ID 4, 3, 2, 5, 1, 0.

To look for first operation executed, we start from the Root (ID 0), going down its child operation ID 1. ID 1 has two child operations, ID2 and ID5, as shown in their indentation. From here, we follow ID 2 until no child nodes are found which in this case ends at ID 4. The flow then goes up its parent operations, ID 4, 3, 2 and visit the unvisited child operation ID 5, go back up again because there are no more child operations to ID1, and finally to ID 0. The data

flow sequence goes as follows ID 4 > 3 > 2 > 5 > 1 > 0. Next, we will try to read and interpret the reads line by line.

Row 4 - Fast Dual and expected number of rows is 1. In this operation, the optimizer executes a query from the "dual" table which is a special temporary table that only returns 1 value.

Row 3 - Index Range Scan, as mentioned on the access path section, this operation can return multiple RowIDs that match the predicate expression, in our case the table being accessed is the USER_ALLOWED_SITE_PK which fetches the user id and the sites it has permission to view.

Row 2 - VIEW operation creates an intermediate result set from the index range scan results which will be used compared to ID 5' results.

Row 5 - Table Access Full on INVENTORY_PART_TAB, expected number of rows returned 298,532. We are access all the rows on the INVENTORY_PART_TAB as we did not specify any WHERE clause in the query.

Row 1 - HASH JOIN RIGHT SEMI - As explained on the execution plan join types, a Hash Join hashes the smallest table of the two tables on the JOIN operation (In this case the Site results in VW_SQ1), then the same hashing is applied on the second table (INVENTORY_PART_TAB). The operation will then compare matching hash results from Table 1 and Table 2 and returns those results. Although we did not specify a Join operation anywhere, this is IFS's way of checking whether the user fetching the data has access to the site information on inventory part.

Row 0 - SELECT STATEMENT, expected Rows 54,631. This row operation is the summary of all the operations that have occurred below it. The expected number of rows in this case is just an estimation based on the database statistics. The results might differ for each user due to permission differences.

2.3.5 Interpreting the execution plan results

For every row on the Execution Plan is an operation. Each operation can be either an Access method, where a table, view, or join result is being accessed in a certain way, or a Join Operation, where two results from an operation are joined together. In this sub chapter, we will be looking into the different operations that can be found in an Execution Plan such as Access Methods and Join Operations.

2.3.6 Execution Plan Table Access Methods/Path

An Access Path in an Execution plan is a method of which data is retrieved from a row source. Row sources can be in a form of a Table, View, or a result of a Join Operation combining two

tables into one (Optimizer Access Paths, 2013). Although there are quite many Access Methods, it can be broken down into two categories, Full Table Scan, and Index Scans.

2.3.6.1 Full Table Scan (FTS)

A full table scan in an execution plan will read all rows from a table and filters the rows that do not match the selection criteria. This is typically done when the optimizer cannot use other access paths such an Index, or if the result of doing a full table scan will yield lower cost (Optimizer Access Paths, 2013). Below is a table of reasons when the optimizer will perform a Full Table Scan.

Reason	Explanation
No index exists.	When an Index does not exist, a Full Table Scan is used.
The query predicate applies a function to the indexed column.	Unless the index is also a function-based index, when an indexed column is put into a function, the ability to use the index can be ignored. Ex. UPPER(column_name) = 'TEST'
A SELECT COUNT(*)query is issued, and an index exists, but the indexed column contains nulls.	The optimizer cannot use the index to count the number of table rows because the index cannot contain null entries.
The query predicate does not use the leading edge of a B-tree index.	This occurs when for example, an index might exist on employees table on columns first_name and last_name. When a query is executed with the predicate is WHERE last_name = 'RAZON, the optimizer may not choose an index because column first_name is not in the predicate. This can however result in an Index Skip Scan instead if indexes are available on the two columns.
The query is unselective.	An unselective query is one that does not specify enough access predicates to the WHERE clause or JOIN operation. This would make the optimizer think that it requires most of the blocks in a table to be read leading to a Full Table Scan.

The table statistics are stale.	Stale statistics occur when a table that had few rows had grown significantly. If the statistics are not up to date and is using old statistic data, the optimizer might think that performing a full table scan is more efficient than an index access.
The table is small.	The optimizer might decide on a full table scan if the table contains very few rows below a threshold. Also, if the cost of performing a full table scan is lower than that of an index access, a full table scan will be used due to the Cost Based Optimization.
The table has a high degree of parallelism.	The optimizer will opt for a Full Table Scan over Indexes when there is a high degree of parallelism.
The query uses a full table scan hint.	The use of a hint FULL(table_name) forces the optimizer to use a full table scan.

Table 1: Optimizers reasons for choosing an FTS (Optimizer Access Paths, 2013)

Full Table Scans are not necessarily a bad thing in an execution plan, as mentioned that if the table is small enough, the optimizer might opt for an FTS rather than use an Index as it would result in a lower cost. Knowing and understanding the properties of the table a developer is working with is important to understand the optimizers decision to use an FTS is justified from the number of rows.

2.3.6.2 Table Access by Row ID

Considered to be the fastest way of accessing a single row of data, a Table Access by Row ID is a method of access by which a Row ID is supplied to point to the exact physical address location of the row in the database. A Row ID lookup is faster than a Primary Key lookup and is unique to every row on a table and the entire database (Optimizer Access Paths, 2013).

	ROWID	ROWNUM	EMPNO	ENAME
▶ 1	AAAQ+jAAEAAAAAcAAA	1	7999	jimzhang
2	AAAQ+jAAEAAAAAeAAA	2	7369	SMITH
3	AAAQ+jAAEAAAAAeAAB	3	7499	ALLEN
4	AAAQ+jAAEAAAAAeAAC	4	7521	WARD
5	AAAQ+jAAEAAAAAeAAD	5	7566	JONES
6	AAAQ+jAAEAAAAAeAAE	6	7654	MARTIN
7	AAAQ+jAAEAAAAAeAAF	7	7698	BLAKE
8	AAAQ+jAAEAAAAAeAAG	8	7782	CLARK
9	AAAQ+jAAEAAAAAeAAH	9	7788	SCOTT

Rowid and Rownum in SQL

Figure 14: Row IDs in a table (What is ROWID and ROWNUM in SQL?, 2019)

2.3.6.3 Index Unique Scan

An Index Unique Scan occurs when all the indexed columns of a table are referenced in the query predicate such as the WHERE clause using an equality operator (Srivastava, n.d.). This access method scans an indexed table and fetches a single RowID to be used to retrieve the row data (Optimizer Access Paths, 2013).

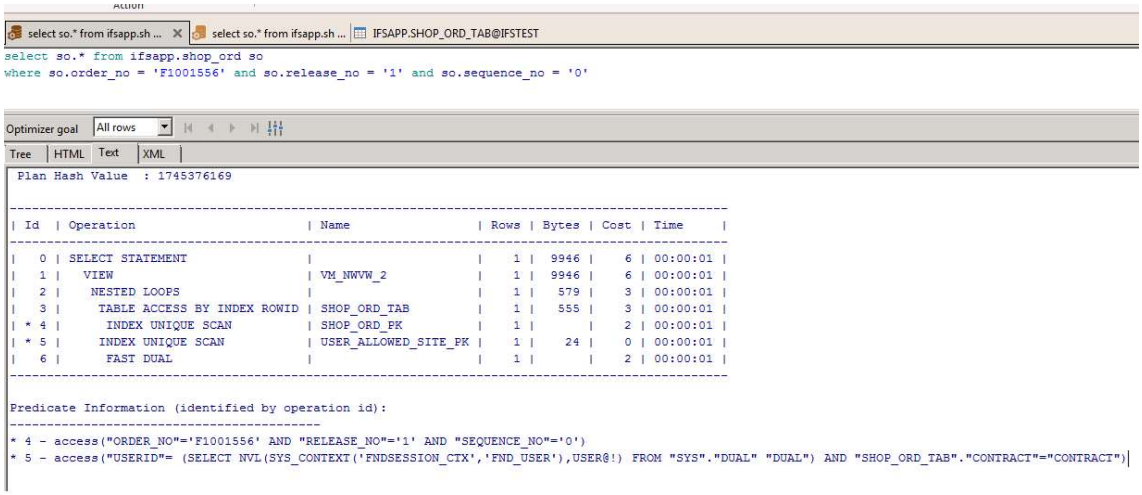


Figure 15: Index Unique Scan in an Execution Plan

Image above shows an Index Unique Scan in the query's execution plan (ID4) for fetching the Shop Order information. The access method was chosen by the optimizer because all the indexed columns for shop_ord_pk was supplied in the WHERE clause of the query using an equality operator. See image below of the Indexed Columns for SHOP_ORD_PK.

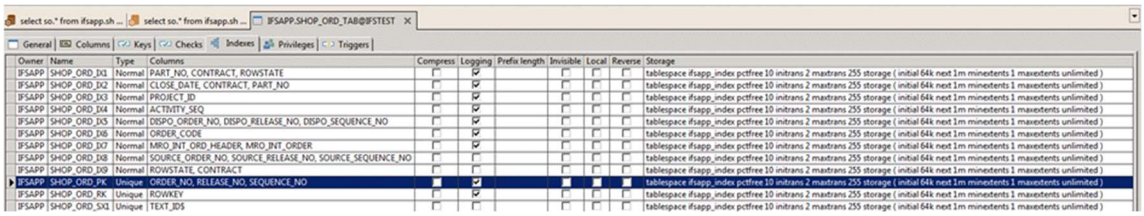


Figure 16: Indexes for the SHOP_ORD_PK Columns ORDER_NO, RELEASE_NO, SEQUENCE_NO

2.3.6.4 Index Range Scan

Compared to an Index Unique Scan, an Index Range Scan can return multiple RowID values and occurs when in the predicate expression(s) in the WHERE clause has >,<, >=, <= conditions on the indexed columns(WHERE column > 1000) or when not all of the indexed columns are referenced in the WHERE clause (Srivastava, n.d.).

The image below shows an Index Range Scan in the query's execution plan (ID5) for fetching the Shop Order information. By not supplying the release no and sequence no, the optimizer instead chooses an Index Range Scan as there can be multiple orders with varying release and sequence numbers. This value is shown in the Rows (Cardinality) column as the optimizer estimates the expected rows to be 2.

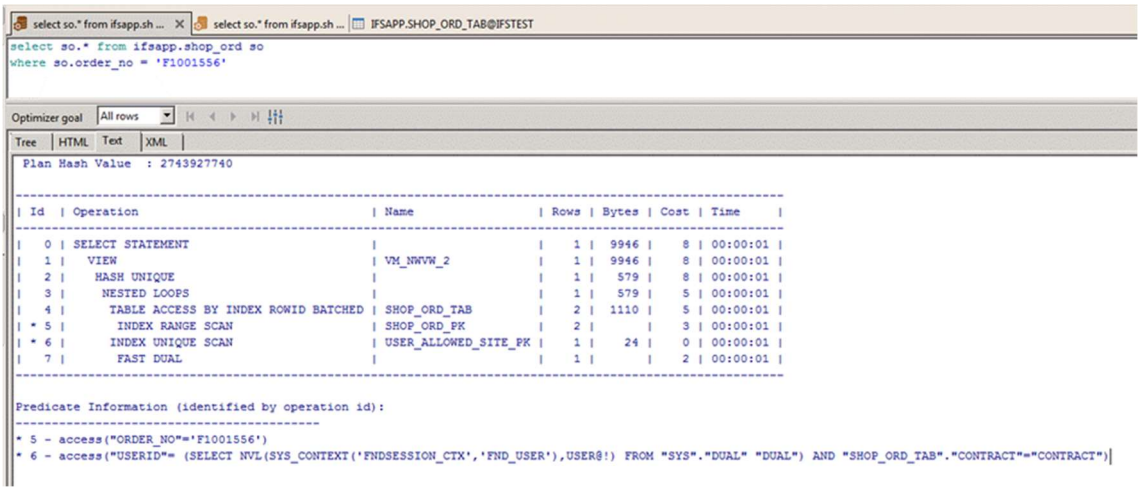


Figure 17: Index Range Scan in an Execution Plan

2.3.6.5 Index Skip Scan

When the first in a multi-column index is not supplied in the WHERE clause of a query, instead it is supplied its leading index column(s), then the optimizer might opt to use an Index Skip Scan if the first column contains very few distinct values. By its nature, this type of index scan is not as efficient compared to other index scans as it requires to traverse the indexes multiple times (Srivastava, n.d.).

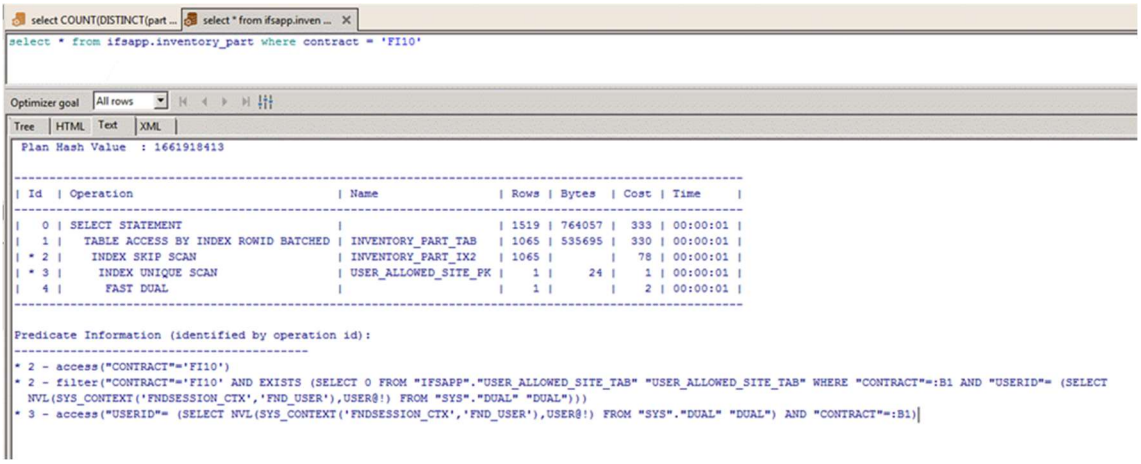


Figure 18: Index Skip Scan in an Execution Plan

In the example above, the inventory_part_tab is indexed using the part_no and contract. In the query, only the contract was given in the WHERE clause, skipping the first index column. Due to the low distinct rows for the first columm, an Index Skip Scan was chosen by the optimizer.

2.3.6.6 Full Index Scan

In an Index Full Scan, the entire index table is read in its original sort order. This eliminates the need for a sort operation due to the nature of indexes already being pre-sorted. The optimiser chooses this type of access method when: First, any index columns are referenced in the WHERE clause of an SQL statement. Second, the selected columns in the query are all part of the index, and that one indexed column is not null. Lastly, when an ORDER BY keyword is specified to a column in the query that is non-nullable, meaning a column that cannot be empty such as a Primary Key (Balasubramanian, n.d.).

2.3.6.7 Fast Full Index Scan

A Fast Full Index Scan is a replacement for a Full Table Scan, wherein it reads all the blocks in an index in an unsorted manner. This is chosen by the optimizer when the indexed columns selected by the query are all present in the index table itself (Optimizer Access Paths, 2013).

2.3.6.8 Index Join

An Index Join Scan occurs when the optimizer is working with a table that contains multiple indexes and that the query SELECT have all the columns it needs in those indexes. This way, the optimizer will find it more efficient to search multiple indexes and Hash Join them together to get the result set. In this access method, the optimizer will only have to work with the Index table rather than accessing the table itself (Colgan, 2021).

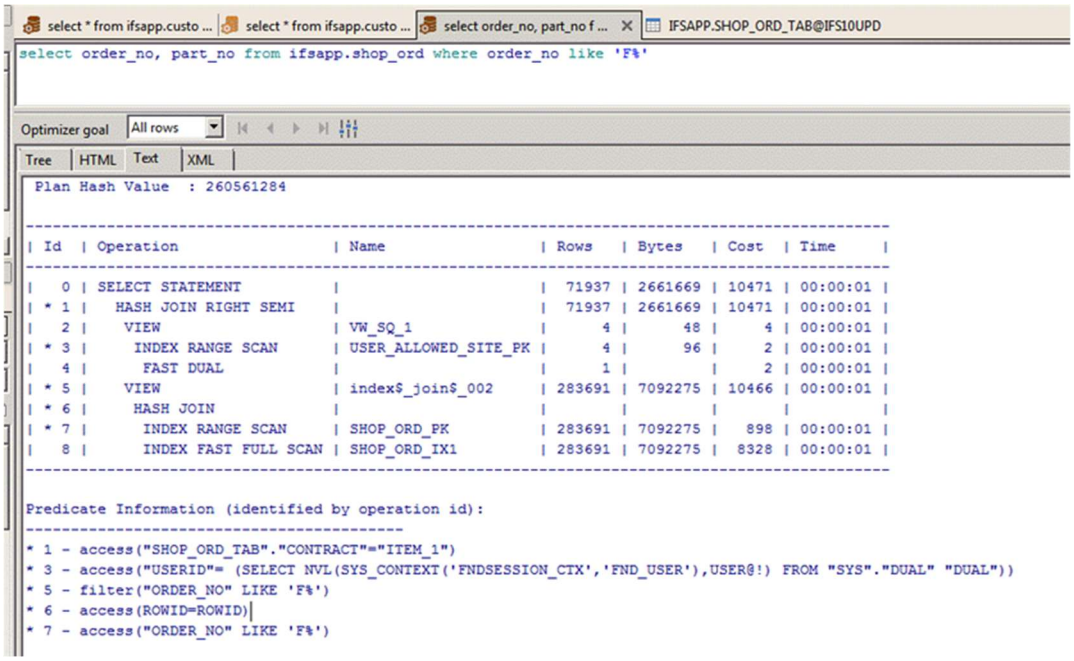


Figure 19: Index Join Scan in an Execution Plan

As mentioned, that an Index Join Scan is a Hash Join of indexes, the image above shows an Index Join Scan where in 2 columns (order_no, part_no) in the SQL query are indexes on SHOP_ORD_PK (order_no) and SHOP_ORD_IX1(part_no).

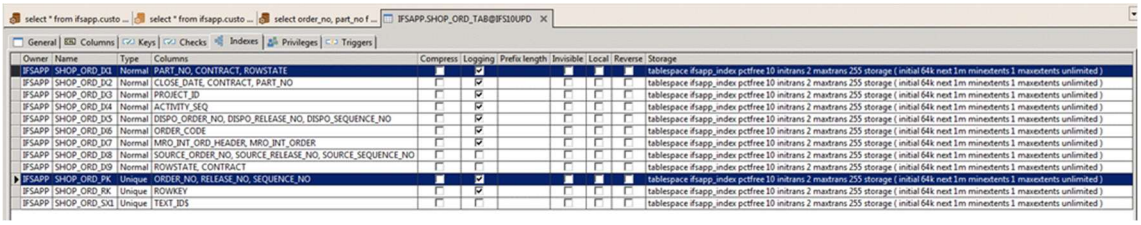


Figure 20: SHOP_ORD schema showing the indexes for SHOP_ORD_IX1 and SHOP_ORD_PK

2.3.7 Execution Plan Join Methods

Join Operations in an Execution Plan is a result of two row sources combined based on a similar column that they both have. The row source can be any combination of table, view, or other join operations.

2.3.7.1 Nested Loops

A Nested Loop is a type of join method which functions like a nested for loop, as seen in the image below. Every row in the outer table will be matched against all the rows on the inner table. This join method is a proper operation when the optimizer is working with a few rows and an efficient access path such as an index in the inner table, as the inner table will be

read over and over for each row from the outer loop matching the WHERE clause (Colgan, 2021).

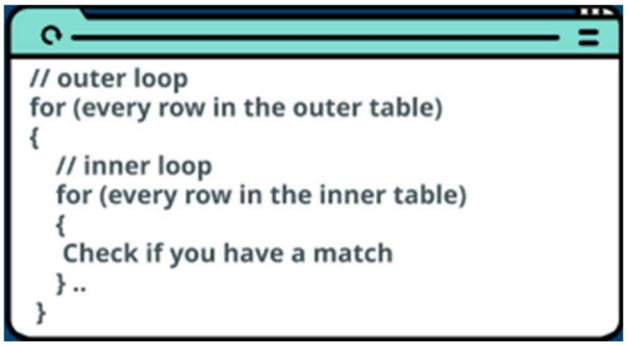


Figure 21: Illustration of a Nested Loop (Colgan, 2021)

2.3.7.2 Hash Join

In working with larger tables, the optimizer might opt to use a Hash Join. A Hash Join works by building a hash table in memory using the smaller table in the join operation. Once a hash table has been made using the smaller table, the same hashing function will be applied to the second table, which the optimizer will then compare the hashed values on both tables to look for matches (Colgan, 2021).

2.3.7.3 Sort Merge Join

The Sort Merge join is a twostep process of sorting and merging tables. This operation is done when either or both inputs on the join columns are already sorted, in most cases when the column(s) are indexes. The first step takes the first-row source then sorts it by the join column(s), then it does the same on the second table. Next, the matching results are merged in the original order (Colgan, 2021).

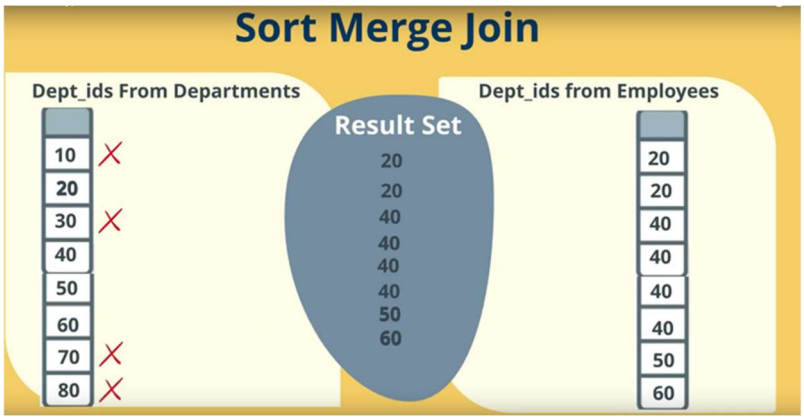


Figure 22: Sort Merge Join Illustration (Colgan, 2021)

3 Application of RCA in tuning SQL using the Execution Plan

This chapter will describe the application of the tools and methods described in chapter 2.0 to tune an SQL query.

3.1 Define the problem or areas of improvement

The first step in RCA is to define the problem or areas needing of improvement. In the context of SQL tuning, is to identify a high-load SQL statement. For the purposes of this thesis, a suboptimal query will be purposefully created to showcase different issues that can be identified in an execution plan and demonstrate how the problem can be improved on.

```
SELECT *
FROM ifsapp.shipment_line sl, ifsapp.shop_ord so, ifsapp.inventory_part ip
WHERE UPPER(ip.description) LIKE '%RAL%' AND sl.source_ref1 = so.order_no
and so.order_no ='F1000478'
```

Figure 23: Suboptimal SQL query

To demonstrate a suboptimal performing query, the SQL above will be used as an example. The query functions by selecting all the rows (SELECT *) on the tables specified on the FROM clause (ifsapp.shipment_line, ifsapp.shop_ord, ifsapp.inventory_part), and search and filters data on the WHERE clause. In the WHERE clause, the query has three conditions that should all be true to return a result. First, the query searches for a matching row on the Shop Ord table provided the order number (so.order_no = ‘F1000478’). Second, it checks whether the provided shop order number matches a shipment line on the source_ref1 column (sl.source_ref = so.order_no). Lastly, it filters the returned results of a shipment line that contains an inventory part description using the RAL colour standard (UPPER(ip.description) LIKE ‘%RAL%’), the UPPER function converts the result in ip.description to all upper case characters.

3.2 Assemble as much data and inputs as possible

The second step in RCA is the data gathering phase or gathering SQL performance related data. For that, the Execution Plan of the SQL in step 1 and the available indexes of the tables used by the plan will be gathered.

To gather the plan, the query in Step 1 is ran on the Explain Plan window as instructed on the previous chapter on how to view the execution plan in PL/SQL Developer Tool. Any access predicate statements on the WHERE clause will be replaced with a bind variable to act as a temporary place holder. In the case of the query, the so.order_no ='F1000478' will be replaced with so.order_no =:order_no. Running the SQL in the Explain Plan window will result with the plan below.

```
SELECT *
FROM ifsapp.shipment_line sl, ifsapp.shop_ord so, ifsapp.inventory_part ip
WHERE UPPER(ip.description) like '%RAL%' and sl.source_ref1 = so.order_no and so.order_no =:order_no
```

Optimizer goal: All rows

Tree | HTML | Text | XML

Description	Id	Object owner	Object name	Cost	Cardinality	Bytes	Time
SELECT STATEMENT, GOAL = ALL_ROWS	0			5 853	296	363 784	1
HASH JOIN RIGHT SEMI	1			5 853	296	363 784	1
VIEW	2	SYS	VW_SQ_1	4	4	48	1
INDEX RANGE SCAN	3	IFSAPP	USER_ALLOWED_SITE_PK	2	4	96	1
FAST DUAL	4			2	1		1
MERGE JOIN CARTESIAN	5			5 849	1 576	1 917 992	1
NESTED LOOPS SEMI	6			12	1	714	1
MERGE JOIN CARTESIAN	7			9	1	712	1
TABLE ACCESS BY INDEX ROWID BATCHED	8	IFSAPP	SHIPMENT_LINE_TAB	5	1	156	1
INDEX RANGE SCAN	9	IFSAPP	SHIPMENT_LINE_IX1	3	4		1
BUFFER SORT	10			4	2	1 112	1
TABLE ACCESS BY INDEX ROWID BATCHED	11	IFSAPP	SHOP_ORD_TAB	4	2	1 112	1
INDEX RANGE SCAN	12	IFSAPP	SHOP_ORD_PK	2	2		1
VIEW PUSHED PREDICATE	13	SYS	VW_SQ_2	3	1	2	1
INDEX UNIQUE SCAN	14	IFSAPP	USER_ALLOWED_SITE_PK	1	1	24	1
FAST DUAL	15			2	1		1
BUFFER SORT	16			5 846	14 927	7 508 281	1
TABLE ACCESS FULL	17	IFSAPP	INVENTORY_PART_TAB	5 837	14 927	7 508 281	1

Figure 24: Suboptimal Execution Plan

Next, is to identify the available indexes for the tables being used. The tables used by the plan can be seen on the Object name column. Because the query is working with the tables Shipment Line, Shop Order, and Inventory Part, the attention can be focused on the Object names that are like the tables in the WHERE clause, namely, ID 8, 9, 11, 12, and 17. Other tables such as ID 2 and 13 are views and act as a temporary table for data, while ID 3 and 14 are more filters checks done IFS based on the permission of a user. Hovering the mouse over the object name and pressing the RMB, then selecting view will show the structure of the table. For this thesis, the primary focus will only be on the Indexes Tab.

General | Columns | Keys | Checks | Indexes | Privileges | Triggers

Owner	Name	Type	Columns	Compress	Logging	Prefix length	Invisible	Local	Reverse	Storage
IFSAPP	INVENTORY_PART_IX1	Normal	INVOICE_CONSIDERATION		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 maxextents unlimited)
IFSAPP	INVENTORY_PART_IX2	Normal	SUPERSEDES, CONTRACT							tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 maxextents unlimited)
IFSAPP	INVENTORY_PART_PK	Unique	PART_NO, CONTRACT		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 maxextents unlimited)
IFSAPP	INVENTORY_PART_RK	Unique	ROWKEY		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 maxextents unlimited)
IFSAPP	INVENTORY_PART_SX1	Unique	TEXT_ID$		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 maxextents unlimited)

Figure 25: Inventory Part Table Indexes

General | Columns | Keys | Checks | Indexes | Privileges | Triggers

Owner	Name	Type	Columns	Compress	Logging	Prefix length	Invisible	Local	Reverse	Storage
IFSAPP	SHIPMENT_LINE_IX1	Normal	SOURCE_REF1, SOURCE_REF2, SOURCE_REF3, SOURCE_REF4, SOURCE_REF_TYPE							tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next
IFSAPP	SHIPMENT_LINE_PK	Unique	SHIPMENT_ID, SHIPMENT_LINE_NO		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next
IFSAPP	SHIPMENT_LINE_RK	Unique	ROWKEY		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next
IFSAPP	SHIPMENT_LINE_UX1	Unique	SHIPMENT_ID, SOURCE_REF1, SOURCE_REF2, SOURCE_REF3, SOURCE_REF4, SOURCE_							tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next

Figure 26: Shipment Line Indexes

General | Columns | Keys | Checks | Indexes | Privileges | Triggers

Owner	Name	Type	Columns	Compress	Logging	Prefix length	Invisible	Local	Reverse	Storage
IFSAPP	SHOP_ORD_IX1	Normal	PART_NO, CONTRACT, ROWSTATE		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_IX2	Normal	CLOSE_DATE, CONTRACT, PART_NO		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_IX3	Normal	PROJECT_ID		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_IX4	Normal	ACTIVITY_SEQ		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_IX5	Normal	DISPO_ORDER_NO, DISPO_RELEASE_NO, DISPO_SEQUENCE_NO		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_IX6	Normal	ORDER_CODE		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_IX7	Normal	MRO_INT_ORD_HEADER, MRO_INT_ORDER		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_IX8	Normal	SOURCE_ORDER_NO, SOURCE_RELEASE_NO, SOURCE_SEQUENCE_NO							tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_IX9	Normal	ROWSTATE, CONTRACT							tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_PK	Unique	ORDER_NO, RELEASE_NO, SEQUENCE_NO		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_RK	Unique	ROWKEY		☑					tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma
IFSAPP	SHOP_ORD_SX1	Unique	TEXT_ID$							tablespace ifsapp_index pctfree 10 initrans 2 maxtrans 255 storage (initial 64k next 1m minextents 1 ma

Figure 27: Shop Order Indexes

With the Execution Plan and Indexes gathered, the results can then be used to check for possible improvements.

3.3 Locate the causes

The third step is locating the cause(s) of the problem. Using the plan and available indexes gathered on step two, the type of operations performed can be identified on the plan and compared to the SQL to determine the problem.

Without reading the entire flow of the plan, a few red flags can already be identified on the plan while comparing its SQL query.

- TABLE ACCESS FULL on ID 17, Inventory Part table. The cardinality estimate is 14,927 rows being returned.
  A table access full is not necessarily a bad thing in an Execution Plan if the tables are small enough then the cost of performing the operation is going to be lower than that of an index access, but that is not that case for this SQL. The Inventory Part table contains a large row count as it contains data for the products at Teknos. Also, the SQL query does not provide any access predicate such as an Inventory Part ID and Site information, as shown with the available indexes for Inventory Part in step 2. The use of a double wildcard is also an expensive operation because each row needs to be checked for any matching patterns with RAL, this solution can also return an incorrect result as some Teknos product names might use the letters ral. There should be other available columns that can provide the RAL color code information.

- MERGE JOIN CARTESIAN on ID 5 and 7. ID 7, is an inner cartesian join operation, joining tables SHIPMENT_LINE_TAB and SHOP_ORD_TAB. The result from ID 7's cartesian join is then sent to ID 6 to be compared against the USER_ALLOWED_SITE_PK results to check for site access permissions. ID 5, is an outer cartesian join which then joins results from the inner cartesian join to the buffer sort result of INVENTORY_PART_TAB. The overall result is a join of 3 tables along with one table that performed a Full Table Scan.

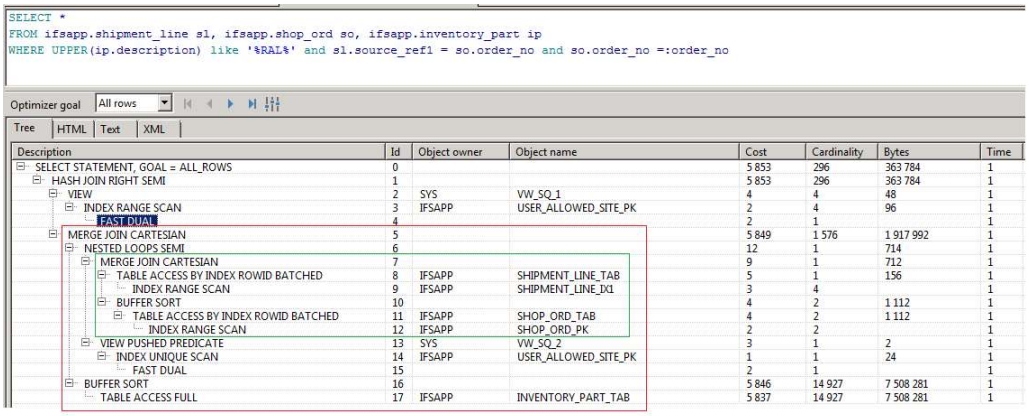


Figure 28: Inner (Green Box) and Outer (Red Box) Cartesian Joins

Cartesian Joins are known to be expensive design operations in a SQL. This occurs when the SQL did not specify a JOIN operation of tables or insufficient parameters supplied to the WHERE clause. This results in a large table whose rows are all join together creating a cartesian product (1). The JOIN operation should be specified on the FROM clause using columns that match two certain tables namely, the Shipment Line and Shop Order.

- SELECT * in the SQL. Using the SELECT * selects all columns on the three tables specified on the FROM clause. A rule of thumb is to only select the amount of columns needed as it can reduce the amount of cost of the plan as well as possibly improve the selection process of the table access methods if the optimizer can fetch the rows in the Index tables itself.

Although not considered as a red flag in the plan. Another possible room for improvement is the INDEX RANGE SCAN on ID9 and ID12. An Index Range Scan is a good access operation as it uses the available indexes and can read rows quickly and efficiently, but it is possible to turn it into an INDEX UNIQUE SCAN to improve the selectivity of the query to only 1 row if there is an available Unique type of index in the index table.

After analysing the plan and identified possible performance causes, the following goals can then be set as solutions to the identified problem.

- FULL TABLE SCAN - Eliminate the Full Table Scan and use the available indexes and find another column for the colour code.

- MERGE JOIN CARTESIAN - Specify a JOIN operation between the Shipment Line and Shop Order and Inventory Part table to eliminate the Cartesian Product result.

- SELECT * - Select only the needed column(s), instead of a SELECT *.

- INDEX RANGE SCAN - Improve on the Index Range Scan if the tables have unique indexes available.

In the next step, the focus is to apply these changes to the SQL query.

3.4 Find Corrective and Preventive Solutions

Having identified four problems to fix, the fourth step in RCA is to find and implement corrective and preventive solutions to the SQL.

Starting with the first problem of a FULL TABLE SCAN on ID17 on the Inventory Part table. The goal is to eliminate the Full Table Scan and change the filter description to a more specific column. Looking into Figure 25 of the Inventory Part indexes, an Index of type Unique is available on the third row when the PART_NO and CONTRACT is supplied as access predicates. The Part No and Contract can be supplied in the WHERE clause or via a JOIN operation because the Shop Order tables contains a column called Part No and Contract. The wildcard operation to filter for the RAL colour code is to be changed to a custom field column called CF$_COLOR_CODE on the custom field table of the Inventory Part.

Second problem, MERGE JOIN CARTESIAN on ID 5, and 7. To remove the cartesian operation, a JOIN must be specified to the three tables along with connecting columns. Shipment Line and Shop Order both have columns that can be joined. For Shop Order, these are columns ORDER_NO, RELEASE_NO and SEQUENCE_NO, joined in the same order to SOURCE_REF1, SOURCE_REF2, SOURCE_REF3 for Shipment Line. The columns are the same type of data but use different terminologies in a different context. For the Inventory Part table, Part no and Contract is also available on Shop Order. The Inventory Part columns PART_NO and CONTRACT can be joined to Shop Orders PART_NO and CONTRACT column.

Third problem, SELECT *. Properly defining which columns are needed and will be used is good practice in writing effective SQL. For this problem, only one column will be selected instead of the SELECT *.

Lastly, an Index Range Scan. As mentioned earlier that an index range scan is a good access method, but it can still be improved on. To solve this issue is to use the available unique indexes for the tables mentioned in Step 2.

Having identified possible solutions to the problem, step 5 applies the changes to the SQL query and explains why the solutions are implemented.

3.5 Create Actionable strategies to implement the solution

Using all the solutions mentioned in step 4. The SQL query can be re-written as follow:

```
SELECT sl.shipment_id
FROM ifsapp.shipment_line sl
INNER JOIN ifsapp.shop_ord so ON sl.source_ref1 = so.order_no AND
sl.source_ref2 = so.release_no AND sl.source_ref3 = so.sequence_no
INNER JOIN ifsapp.inventory_part_cfv ip ON ip.part_no = so.part_no AND
ip.contract = so.contract
WHERE so.order_no =:order_no AND so.release_no =:release_no AND
so.sequence_no =:sequence_no AND ip.CF$_COLOR_CODE LIKE 'RAL%'
```

Figure 29: Rewritten SQL query

Eliminating the Full Table Scan. Using the available unique indexes on Inventory Part, the table is now joined to Shop Orders PART_NO and CONTRACT column (INNER JOIN ifsapp.inventory_part_cfv ip ON ip.part_no = so.part_no AND ip.contract = so.contract). Also, the double wildcard search on the description column has been replaced with a single wildcard search on CF$_COLOR_CODE, this column contains the proper color code standards data from the ifsapp.inventory_part_cfv table, which is the inventory part table containing all custom fields separate from the original table.

Fixing the Cartesian Joins. The three tables all have columns related to each other, therefore a join of the three is possible in the following order, Shipment Line joins to Shop Order and Shop Order joins to Inventory Part (ifsapp.shipment_line sl INNER JOIN ifsapp.shop_ord so ON sl.source_ref1 = so.order_no AND sl.source_ref2 = so.release_no AND sl.source_ref3 = so.sequence_no INNER JOIN ifsapp.inventory_part_cfv ip ON ip.part_no = so.part_no AND ip.contract = so.contract).

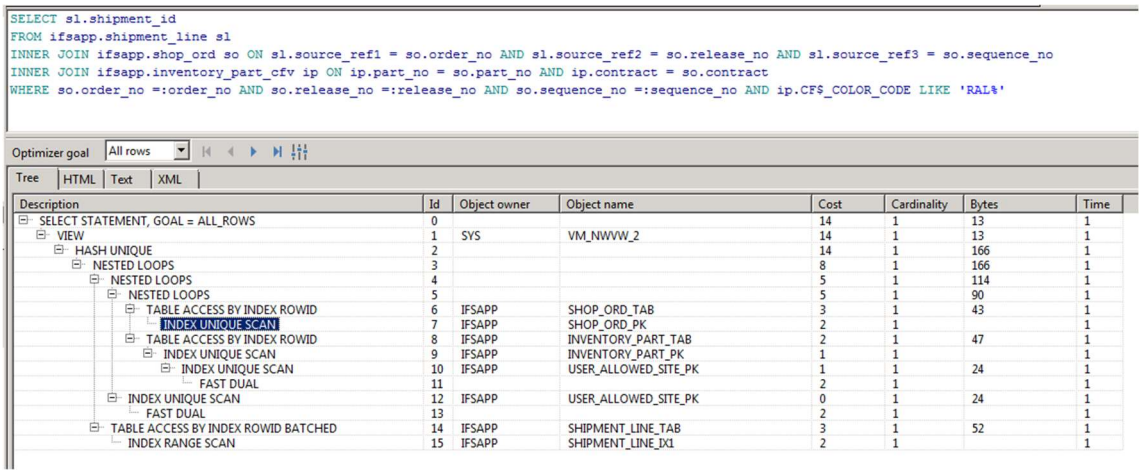
Select * columns. For this problem, the Shipment Lines shipment_id column is selected. (SELECT sl.shipment_id)

Improving on the Index Range Scan. With the tables all joined together to their corresponding index columns. The Shop Order Index Range scan on ID 12 should result in an Index Unique Scan because all index unique columns Order, Release, and Sequence Numbers are joined to Shipment Line and are supplied in the WHERE clause (WHERE so.order_no =:order_no AND so.release_no =:release_no AND so.sequence_no =:sequence_no). For the Shipment Line, the operation will result in an Index Skip Scan or Index Range scan because the conditions for a Unique index were not supplied and that the columns SHIPMENT_ID and SHIPMENT_LINE_NO are only available on Shipment lines and cannot be joined to Shop Order.

With the new SQL query using more efficient indexes, joins, and selecting only needed columns. The SQL should return a more efficient Execution Plan utilizing Index Unique Scans, lower cardinality estimates and overall cost.

3.6 Monitor the solution and confirm if it works

The last step in RCA in SQL tuning is to confirm the solutions effectiveness. For the last step, the new SQL code is rerun on the Explain Plan window to check and confirm the new Execution Plan result.



```
SELECT sl.shipment_id
FROM ifsapp.shipment_line sl
INNER JOIN ifsapp.shop_ord so ON sl.source_ref1 = so.order_no AND sl.source_ref2 = so.release_no AND sl.source_ref3 = so.sequence_no
INNER JOIN ifsapp.inventory_part_cfv ip ON ip.part_no = so.part_no AND ip.contract = so.contract
WHERE so.order_no =:order_no AND so.release_no =:release_no AND so.sequence_no =:sequence_no AND ip.CF$_COLOR_CODE LIKE 'RAL%'
```

Optimizer goal: All rows

Tree | HTML | Text | XML

Description	Id	Object owner	Object name	Cost	Cardinality	Bytes	Time
SELECT STATEMENT, GOAL = ALL_ROWS	0			14	1	13	1
VIEW	1	SYS	VM_NWVW_2	14	1	13	1
HASH UNIQUE	2			14	1	166	1
NESTED LOOPS	3			8	1	166	1
NESTED LOOPS	4			5	1	114	1
NESTED LOOPS	5			5	1	90	1
TABLE ACCESS BY INDEX ROWID	6	IFSAPP	SHOP_ORD_TAB	3	1	43	1
INDEX UNIQUE SCAN	7	IFSAPP	SHOP_ORD_PK	2	1		1
TABLE ACCESS BY INDEX ROWID	8	IFSAPP	INVENTORY_PART_TAB	2	1	47	1
INDEX UNIQUE SCAN	9	IFSAPP	INVENTORY_PART_PK	1	1		1
INDEX UNIQUE SCAN	10	IFSAPP	USER_ALLOWED_SITE_PK	1	1	24	1
FAST DUAL	11			2	1		1
INDEX UNIQUE SCAN	12	IFSAPP	USER_ALLOWED_SITE_PK	0	1	24	1
FAST DUAL	13			2	1		1
TABLE ACCESS BY INDEX ROWID BATCHED	14	IFSAPP	SHIPMENT_LINE_TAB	3	1	52	1
INDEX RANGE SCAN	15	IFSAPP	SHIPMENT_LINE_IX1	2	1		1

Figure 30: New Execution Plan for the SQL

By implementing the solutions to the query such as utilizing the available index, using join operations, adding more access predicates, using columns that are more suited for the operation, and selecting only the column needed. The resulting Execution Plan is using more efficient access paths such as INDEX UNIQUE SCANs and INDEX RANGE SCAN while having lower cost and cardinality estimates. The cartesian joins are gone after using the join operation on the three tables using their indexes. The Inventory Part table is now being accessed via an INDEX UNIQUE SCAN rather than a FULL TABLE SCAN and that the filter predicate is now using a more efficient column that only needs one wildcard operation. The Shop Order table improved from an INDEX RANGE SCAN to an INDEX UNIQUE SCAN after joining all three columns in the SHOP_ORD_PK and providing the data in the WHERE clause. The cost and cardinality estimates have lowered because the query is only selecting 1 column and is now more selective with the added access predicates supported by the INDEX UNIQUE SCAN result.

4 Results

The primary goal of this thesis project was to gain an understanding on how the Execution Plan can be utilized to identify SQL query issues and enhance a query's performance using the Execution Plan results. The Optimizer, the software responsible for creating the execution plan, its inner workings was elaborate upon to gain an insight of its decision-making process as well as the factors that affects its decisions. Understanding the optimizer helped gain a base knowledge for the upcoming step in reading the execution plan and interpreting its results using the PL/SQL Developer Tool.

To demonstrate the effectiveness of using the Execution Plan to troubleshoot a query, a suboptimal SQL was purposefully created to show a suboptimal Execution Plan. A 6-step Root Cause Analysis process was then used for the SQL tuning process.

The thesis project was successful in showcasing how the Execution Plan can be used to troubleshoot a suboptimal SQL query and was able to identify problems on the plan which was then leveraged to refactor the code to improve the Execution Plan results. The following issues were identified in the plan using the tools and tuning process:

1. Full Table Scans – Initial Execution Plan showed that the Inventory Part table was being accessed via a Full Table Scan, meaning that all rows were being accessed and read. The operation on the query was estimated to have costed 5 837 in overall resources and have read 14 927 rows with 7.5 Megabytes worth of data (See ID 17 Figure 24). This was refactored in the SQL code by using the available index on inventory part via an INNER JOIN operation on shop order. The result being an Index Unique Scan on inventory part where only row is expected to be read. The operation now estimates to cost 2 in overall resources with only 1 row being read with 47 Bytes of data (See ID 8 Figure 30). Using available indexes in important in an SQL query, it makes accessing the table much quicker instead of going through all the rows on a table. This is especially important when working this tables that have a high number of rows.

2. Cartesian Join Operations – Not specifying a join operation to the tables used resulted in a cartesian join on Figure 24 ID 5, and ID 7. The overall resource cost for the operation on the outer join ID5 was 5 849 with a cardinality estimate of 1 576 rows and 2 Megabytes worth of data. This is the result of the inventory part table being accessed in full as well as insufficient access predicates being provided. By explicitly defining a join operation on all three tables, the cartesian join was changed to an Index Range Scan and Index Unique Scan. Resulting in an overall lower cost as 1 row is expected per to be read at each operation (See Figure 30).

3. SELECT * - Instead of selecting all columns, selecting only what is needed will significantly lower the cost of an operation because less data is being read. This result is noticeable on Figure 24 ID 17, where all rows were read due to the full table scan. The operation read 7.5 Megabytes worth of data. By being more selective and only selecting what is needed, the overall byte cost was reduced in the final execution plan.

4. Index Range Scan – An Index Range Scan is an efficient access method that can read multiple rows, but it can still be improved to an Index Unique Scan which reads only 1 unique row. An Index Unique Scan is possible if an Index of type Unique is available for use in the index table and is supplied in the WHERE clause or a table join. This is

shown on the cardinality estimates on Figure 24 ID 9; cardinality of 4, and ID 12; cardinality of 2. Having joined the shop order to shipment line where all unique indexed columns on shop order where connected, resulted in a plan where the shop order was being accessed via an Index Unique Scan with only 1 row being read. An Index Unique Scan was not possible for shipment line as the shop order did not have any all the columns available for an index unique access, resulting still in an Index Range Scan but with an estimated cardinality of 1, as this operation is a range scan, the cardinality can still go up depending on the shipment lines.

5 Conclusions

With the use of the execution plan, inefficient access methods and high cost were identified on the procedures of an SQL query. The plan results were used to successfully refactor the SQL code to improve the execution plan and performance.

There are numerous ways on how to tune an SQL query's performance. Future recommendations include the use of Tracing tools to get a deeper understanding of actual runtime data as well as to compare the actual execution plan to the estimate execution plan.

The Execution Plan can be used in many more ways than just identifying missing indexes and incorrect access path and join operations. There are many more columns that can be used to show different statistics that could benefit the tuning process.

Other coding practices such as using only PL/SQL function API calls compared to SQL code as well as the impact of switching between PL/SQL functions to SQL code are all areas that require more research to understand their impact in the actual performance.

This thesis project showed the potential of using the execution plan as well as its importance in SQL performance tuning. Overall, the execution plan is only a tool to aid the SQL developer when performance tuning. As I learned throughout this thesis project, the tool will be more effective as the skills of the developer grow.

References

Electronic sources

Teknos. n.d. Teknos history. Accessed 3 September 2021: https://www.teknos.com/company/about-us/our-history/

Teknos. n.d. Teknos key figures. Accessed 3 September 2021: https://www.teknos.com/company/about-us/key-figures/

n.d. Get to know IFS. IFS, p.3. Accessed 5 September 2021: https://www.ifs.com/sitecore/media-library/assets/2015/09/16/11/53/get-to-know-ifs/

Docs.ifs.com. n.d. About Custom Objects. Accessed 5 September 2021: https://docs.ifs.com/techdocs/Foundation1/010_overview/220_user_interface/about_custom_objects/

Docs.ifs.com. n.d. Layered Application Architecture. Accessed 5 September 2021: https://docs.ifs.com/techdocs/Foundation1/010_overview/100_architecture/010_LAA_overview/default.htm

2021. Oracle Database: SQL Tuning Guide. Oracle, pp.29-30, 55-60, 62-63. Accessed 4 September 2021: https://docs.oracle.com/en/database/oracle/oracle-database/21/tgsql/sql-tuning-guide.pdf

Saxon, C., 2020. How to Read an Execution Plan. [online] Oracle Blogs. Accessed 11 September 2021: https://blogs.oracle.com/oraclemagazine/post/how-to-read-an-execution-plan

Techgoeasy. 2019. Oracle Indexes and types of indexes in oracle with example - Techgoeasy. Accessed 11 September 2021: https://techgoeasy.com/oracle-indexes/

Docs.oracle.com. 2013. Optimizer Access Paths. Accessed 12 September 2021: https://docs.oracle.com/database/121/TGSQL/tgsql_optop.htm#TGSQL228

Srivastava, N., n.d. Index Lookup (Unique Scan,Range Scan, Full Scan, Fast Full Scan, Skip Scan). [online] Oracle Database Internal Mechanism. Accessed 16 October 2021: https://databaseinternalmechanism.com/oracle-database-internals/index-lookup-unique-scanrange-scan-full-scan-fast-full-scan-skip-scan/

Balasubramanian, V., n.d. Fast Index Scan, Index Scan, Partition Range Scan and Full Table Scan. DOYENSYS. Accessed 16 October 2021: https://doyensys.com/blogs/fast-index-scan-index-scan-partition-range-scan-and-full-table-scan/

Colgan, M., 2021. Oracle Optimizer Access Methods. Accessed 16 October 2021: https://www.youtube.com/watch?v=jYIjzmYCc0U

Colgan, M., 2021. Explain the Explain Plan: Join Methods. Sqlmaria.com. Accessed 17 October 2021: https://sqlmaria.com/2021/02/02/explain-the-explain-plan-join-methods/

Thebusinessanalystjobdescription.com. 2021. Root Cause Analysis: Process, Techniques, and Best Practices | The Business Analyst Job Description. Accessed 26 October 2021: https://thebusinessanalystjobdescription.com/root-cause-analysis-steps-techniques-and-best-practices/

Okes, D 2009, Root Cause Analysis: The Core of Problem Solving and Corrective Action. Accessed 26 October 2021: ProQuest Ebook Central

Rooney, J. and Vanden Heuvel, L., 2004. Root Cause Analysis for Beginners. QUALITY PROGRESS. Accessed 26 October 2021: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.618.8544&rep=rep1&type=pdf

Gogia, B., 2017. Performance Tuning Basics 1: Selectivity and Cardinality. Expert Oracle. Accessed 26 October 2021: https://expertoracle.com/2017/11/15/db-tuning-basics-1-selectivity-and-cardinality/

SQL WORLD - Parse, Bind, Optimize, Execute. 2019. What is ROWID and ROWNUM in SQL?. Accessed 16 November 2021: https://www.complexsql.com/rowid-rownum/

Allround Automations. 2021. About us - Allround Automations. Accessed 23 October 2021: https://www.allroundautomations.com/about-us/

Docs.oracle.com. 2013. Query Transformations. Accessed 23 October 2021: https://docs.oracle.com/database/121/TGSQL/tgsql_transform.htm#TGSQL206

Figures

Tables