

Advanced Database Manipulation

Prepared by: North Star Consultants (GROUP 1)

Date: November 12, 2023

Submitted by:

Yian Chen

Sean Adam Kagugube

Emmanuella Acheampong

Susan Arzapalo

Introduction

North Star Consultants

North Star Consultants is a leading consultancy dedicated to designing and implementing cutting-edge Database Management Systems (DBMS) for businesses across various industries. With a team of highly skilled database experts and a passion for data optimization, we specialize in helping financial institutions harness the full potential of their data resources.

Mission Statement:

“Our mission is to offer comprehensive DBMS solutions that enable financial institutions to unlock the true value of their data assets and ensure that our clients thrive in a rapidly evolving financial landscape.”

Vision Statement:

“To be the trusted partner of choice for financial institutions seeking excellence in optimizing data storage, retrieval, and analysis, ensuring our clients stay ahead in today's data-driven world”.

Our Services:

- **Data Quality Assurance:** We ensure your data is accurate, consistent, and reliable, eliminating errors and inaccuracies.
- **Data Transformation:** We convert raw data into actionable insights, seamlessly integrating data from multiple sources.
- **Insights for Success:** We provide client-centric insights to empower strategic planning and drive success.
- **Engaging Reporting:** Our visualizations and reports offer clarity and actionable

recommendations.

Why Invest in us?

We have a history of over 100+ successful database implementations and satisfied clients in our 2 years of existence. Our solutions are high in demand and our expertise and commitment to excellence ensure investor confidence in our ability to execute and deliver results.

Customer Background

Horizon Bank is a prominent bank with over 50 branches across the United States. The bank wants to launch a new loan product for its loyal customers (more than 2 years with the bank). In addition, the bank is grappling with soaring marketing expenditures and an alarming drain on resources due to inefficient marketing campaigns. Horizon Bank is in dire need of a database with insights for optimizing their marketing strategies targeting the right customers, reducing expenses, and maximizing the effectiveness of their campaigns to maintain a competitive edge in the financial sector.

Value Proposition

Phase_1: To Build a relational database for managing the data set of the bank and leverage advanced data analytics and predictive modeling

Phase_2: To identify customers who qualify for the loan loyalty product for targeted marketing

Phase_3: To provide customer insights to enable the bank to develop a marketing strategy that minimizes expenses while delivering superior results.

1. VIEW_1: High_Priority_Customers

Description:

The High_Priority_Customers view is designed to identify and categorize customers with balances above \$20,000, classifying them as "High_Priority." This view aligns with the bank's objective of targeting top-priority customers for its new term deposit product that offers an attractive interest rate for customers with high balances.

```
1 # VIEW_1
2 • CREATE VIEW High_Priority_Customers AS
3   WITH P_status AS
4   (
5     SELECT Customer_ID,
6           Balance,
7     CASE
8       WHEN Balance > 20000 THEN "High_Priority"
9       WHEN Balance BETWEEN 5200 AND 20000 THEN "Low_Priority"
10      ELSE "Disqualified"
11    END Priority_Status
12   FROM horizon_bank.customer_accounts
13   )
14   SELECT Customer_ID, First_Name, Last_Name, Priority_Status FROM P_status
15   LEFT JOIN horizon_bank.Customers USING(Customer_ID)
16   WHERE
17     Balance > 20000;
18
19
20
21
```

Action Output

#	Time	Action	Message
14	15:51:47	CREATE VIEW Customers_With_Successful_Contacts AS SELECT C.Customer_I...	0 row(s) affected

The view's benefit to Horizon Bank:

Branch Manager Engagement: The view is crucial for executive management, especially the branch manager, as it identifies top-priority customers. These high-value customers will be personally contacted by the branch manager, ensuring a more personalized and effective marketing approach.

Access for End Users:

The report generated from this view will be shared with executive management, including the

branch manager. To accommodate the non-tech-savvy nature of these users, North Star Consultants will ensure that the report is accessible through user-friendly interfaces or even directly delivered as periodic reports.

VIEW_2: Customers_With_Successful_Contacts

Description:

The Customers_With_Successful_Contacts view identifies customers who have had positive interactions with the bank, offering insights for improved customer relationship management and marketing efficiency.

The screenshot shows a database management tool interface. In the top navigation bar, there are tabs for 'marketing_campaign', 'SQL File 11*', 'REPORT 3_DA*', and 'high_priority_customers'. The main area displays the following SQL code:

```
21
22  -- VIEW_2
23  • CREATE VIEW Customers_With_Successful_Contacts AS
24  SELECT
25      C.Customer_ID,
26      C.First_Name,
27      C.Last_Name
28  FROM
29      horizon_bank.customers AS C
30      INNER JOIN
31      horizon_bank.contacts AS Co USING(Customer_ID)
32  WHERE
33      Co.Previous_Outcome LIKE '%Success%';
34
35
36
37
38
39
40
41
```

The left sidebar shows the 'SCHEMAS' tree with nodes like contacts, customer_accounts, customers, loans, subscriptions, and Views. Under Views, 'customers_with_active_loans', 'customers_with_successful_contacts', and 'high_priority_customers' are listed. The bottom left shows 'Table: customers' and its columns: Customer_ID, MaritalStatus, First_Name, Last_Name, Nationality, Age, Job, and Education.

In the bottom right, the 'Output' pane shows the results of the query execution:

#	Time	Action	Message
14	15:51:47	CREATE VIEW Customers_With_Successful_Contacts AS SELECT C.Customer_I...	0 row(s) affected
15	15:55:47	CRFATF VIFW Customers_With_Active_Loans AS SF1 FCT * FROM horizon_h...	0 row(s) affected

The view's benefit to Horizon Bank:

Enhanced Customer Engagement: This view empowers the contact center team to engage consistently with customers who have a history of successful interactions. It improves loyalty and enables the bank to foster stronger relationships with its customer base.

Access for End Users:

To facilitate easy access for the contact center team, North Star Consultants will implement user-friendly dashboards or reports that highlight customers with successful contacts. These can be integrated into the existing tools used by the contact center team for seamless access.

VIEW_3: Customers_With_Active_Loans

Description:

The Customers_With_Active_Loans view identifies customers with existing loans, aligning with the bank's goal of contacting them to explore opportunities for increased cash flow and eligibility for the new term deposit product. It was realized from phase 2 that only 4 people are categorized as top priority for the bank's new term deposit product due to their huge account balance. Hence by identifying customers with loans (which is a requirement for accessing the new term deposit product), the bank can engage them to ensure they increase their account balance and qualify for the new product.

The screenshot shows the Oracle SQL Developer interface. On the left, the Navigator pane displays the schema structure under 'marketing_campaign'. It lists several tables like 'contacts', 'customer_accounts', 'customers', etc., and views like 'customers_with_active_loans', 'customers_with_successful_contacts', and 'high_priority_customers'. The main panel shows the SQL editor with the following code:

```

35
36      # VIEW_3
37  • CREATE VIEW Customers_With_Active_Loans AS
38  SELECT
39      *
40  FROM
41      horizon_bank.customers AS C
42          JOIN
43      horizon_bank.loans AS L USING(Customer_ID)
44  WHERE
45      (L.Housing_Loan = 'YES' OR L.Personal_Loan = 'YES');
46
47
48
49
50
51
52
53
54
55

```

The 'Output' tab at the bottom shows the execution log:

#	Time	Action	Message
14	15:51:47	CREATE VIEW Customers_With_Active_Loans AS SELECT * FROM horizon_bank.customers AS C	0 row(s) affected
15	15:55:47	CREATE VIEW Customers_With_Successful_Contacts AS SELECT * FROM horizon_bank.customers AS C	0 row(s) affected

The view's benefit to Horizon Bank:

Loan Portfolio Optimization and Customer targeting: This view aids in efficiently managing the loan portfolio by identifying customers with active loans. It also allows the bank to strategically engage with these customers to increase their balances and qualify for the new term deposit product.

Access for End Users:

For the bank's engagement team, including loan officers and contact center representatives, North Star Consultants will implement accessible reporting tools. These tools may include automated reports or dashboards that highlight customers with active loans, ensuring that engagement teams can easily identify and reach out to these customers.

2. Report on Queries Linked to Views for Horizon Bank Optimization

Introduction:

As part of the ongoing initiative to optimize Horizon Bank's marketing strategies and customer

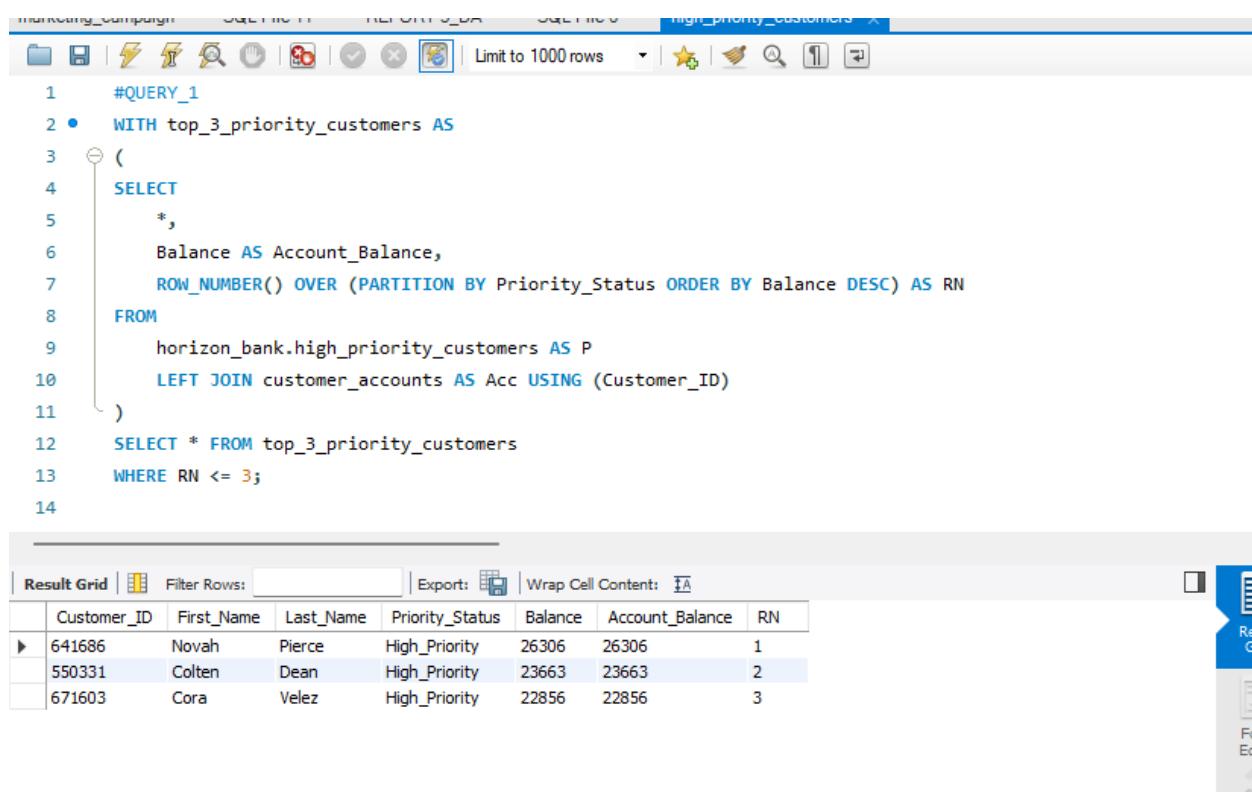
engagement, two queries have been developed based on the previously established views. These queries extract valuable insights from the identified views, providing actionable information for the bank's management.

Query_1: Engagement Strategy for Top Priority Customers

Purpose:

The first query, derived from the High_Priority_Customers view, aims to identify the top three priority customers for engagement by the bank manager. These customers have been categorized as "High_Priority" based on their account balances.

Query Script:



The screenshot shows a SQL editor interface with multiple tabs at the top: 'marketing_campaign', 'SQL FID 1', 'RECENTS_DB', 'SQL FID 0', and 'high_priority_customers'. The 'high_priority_customers' tab is active. Below the tabs is a toolbar with various icons. The main area contains a numbered SQL script:

```
1  #QUERY_1
2  •  WITH top_3_priority_customers AS
3  (
4      SELECT
5          *,
6          Balance AS Account_Balance,
7          ROW_NUMBER() OVER (PARTITION BY Priority_Status ORDER BY Balance DESC) AS RN
8      FROM
9          horizon_bank.high_priority_customers AS P
10     LEFT JOIN customer_accounts AS Acc USING (Customer_ID)
11  )
12  SELECT * FROM top_3_priority_customers
13  WHERE RN <= 3;
14
```

Below the script is a 'Result Grid' table with the following data:

	Customer_ID	First_Name	Last_Name	Priority_Status	Balance	Account_Balance	RN
▶	641686	Novah	Pierce	High_Priority	26306	26306	1
	550331	Colten	Dean	High_Priority	23663	23663	2
	671603	Cora	Velez	High_Priority	22856	22856	3

Interpretation:

The query identifies the top three priority customers, allowing the bank manager to contact three customers each day. The row numbers (RN) provide a sequence for engagement. For instance,

the customer with the highest balance is \$26,306, making them the top-priority customer for immediate engagement.

Suggestions:

To enhance the engagement process, the bank manager could use this information to prioritize their outreach efforts. Additionally, considering the high balance of the top-priority customers, the bank may explore tailored financial products or services to further meet the needs of these valued customers.

Query_2: Analyzing Customer Contact Frequency by Age Group

Purpose:

The second query, derived from the Customers_With_Successful_Contacts view, analyzes the contact frequency of customers, grouped by age. The goal is to provide insights into the effectiveness of customer engagement across different age groups.

Query Script:

```

16      #QUERY_2
17  • ⏪ WITH Contact_Frequency AS (
18      SELECT
19          ROUND(Age / 10) * 10 AS Age_Group,
20          COUNT(*) AS Contact_Count
21      FROM Customers_With_Successful_Contacts AS CS
22      LEFT JOIN customers AS C USING (Customer_ID)
23      GROUP BY ROUND(Age / 10) * 10
24  )
25  SELECT
26      Age_Group,
27      SUM(Contact_Count) OVER (PARTITION BY Age_Group) AS Total_Contacts
28  FROM Contact_Frequency
29  ORDER BY Contact_Count DESC;

```

The screenshot shows a database query editor with the following interface elements:

- Toolbar:** Includes icons for file operations, search, and refresh.
- Query Area:** Displays the SQL code for '#QUERY_2'.
- Result Grid:** Shows the output of the query in a tabular format.

Result Grid Headers:

Age_Group	Total_Contacts
-----------	----------------

Result Grid Data:

40	36
30	35
50	27
60	17
80	6
70	5
20	1

Interpretation:

The query results provide a breakdown of successful customer contacts by age group. The age groups are rounded to the nearest decade, allowing for a comprehensive analysis of contact frequency.

Suggestions:

Targeted Outreach: Recognizing that the age group '40' has the highest contact count, the bank may consider targeted marketing campaigns or personalized offers for customers in this age bracket.

Engagement Strategies: For age groups with lower contact counts, the bank could explore innovative engagement strategies to increase outreach effectiveness.

Customer Segmentation: Further analysis of customer behaviors within each age group could lead to more precise customer segmentation and tailored communication strategies.

3. Indexes: Create and use indexes (on one column and on multiple columns)

Introduction:

Three index sets have been created to increase the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain them. The index and query will be tested on a table sample of 595 rows.

The purpose of the index is to increment the speed of results to identify customers with personal loan, housing_loan, and both to determine their likelihood to sign up for another loan product. In addition, this information will help the bank customer account team to priority which customers they should contact.

Once-Column Index:

```
53 • CREATE INDEX index_personal ON `bank_campaigns` (personal_loan(10));
```

This query creates an index named index_personal on the personal_loan column of the bank_campaign table.

```
55 • CREATE INDEX index_housing ON `bank_campaigns` (housing_loan(10));
```

This query creates an index named index_housing on the personal_housing column of the bank_campaign table.

The (10) after personal_loan and housing_loan suggests that this is a prefix index, which means the index is created on the first 10 characters of the personal_loan and housing_loan columns.

Two-Column Index:

```
CREATE INDEX index_personal_housing ON `bank_campaigns` (personal_loan(10), housing_loan(10));
```

This query creates a composite index named index_personal_housing on two columns: personal_loan and housing_loan of the bank_campaign table. This composite index is useful when queries frequently filter or sort on these columns together.

3.a Speed Comparison for a complex query on a joint of small tables (with and without

index)

To measure and compare the efficiency of the index created, we will test a query in the small table with the index and without the index.

Query without index:

The screenshot shows the MySQL Workbench interface. On the left, the 'Schemas' tree view is open, showing the 'bank_campaign' schema with its tables ('bank_campaign', 'Bank_Campaign', etc.) and indexes ('PRIMARY', 'index_personal', 'index_housing'). A specific index, 'index_housing', is selected. The main pane displays a SQL query and its results. The SQL query is:

```
CREATE INDEX index_personal_housing ON `bank_campaign` (personal_loan(10),housing_loan(10));
```

```
SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans
FROM bank_campaign
GROUP BY housing_loan , personal_loan;
```

The results grid shows the following data:

Housing_Loan	Personal_Loan	Count_of_Loans
no	no	311
yes	no	54
no	yes	218
yes	yes	15

Below the results, the 'Action Output' pane shows the execution log with the following entries:

Time	Action	Response	Duration / Fetch Time
13:32:20	CREATE INDEX index_...	Error Code: 1061. Duplicate key name 'index_personal'	0.0032 sec
13:37:38	SELECT COUNT(*) FR...	Error Code: 1146. Table 'horizon_bank.bank_campaign' doesn't exist	0.0033 sec
13:38:26	SELECT COUNT(*) FR...	1 row(s) returned	0.007 sec / 0.0003...
14:04:31	SELECT Housing_L...	4 row(s) returned	0.0062 sec / 0.0009...
14:06:09	ALTER TABLE 'Horizon...	OK	0.00 sec
14:06:37	SELECT Housing_L...	4 row(s) returned	0.0038 sec / 0.0009...
14:06:39	SELECT Housing_L...	4 row(s) returned	0.0034 sec / 0.0009...
14:06:39	SELECT Housing_L...	4 row(s) returned	0.0033 sec / 0.0009...
14:06:39	SELECT Housing_L...	4 row(s) returned	0.0036 sec / 0.0009...

The speed of the results of the query without index has an average of 0.0027 sec.

Query with Index:

The screenshot shows the MySQL Workbench interface. On the left, the 'Schemas' tree view is expanded to show the 'bank_campaigns' schema, with 'Indexes' selected. The main pane displays a query results grid and a history of executed statements.

```

61 *  SELECT
62   Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans
63   FROM
64   `bank_campaigns`
65   GROUP BY housing_loan , personal_loan;
66

```

The results grid shows the following data:

Housing_Loan	Personal_Loan	Count_of_Loans
yes	no	311
yes	yes	58
no	no	215
no	yes	15

The history pane shows the following log entries:

- 45 14:15:56 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'yes' AND personal_loan = 'no'; 0.0048 sec / 0.00091...
- 46 14:15:57 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'yes' AND personal_loan = 'yes'; 0.0038 sec / 0.00091...
- 47 14:15:58 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'no' AND personal_loan = 'no'; 0.0033 sec / 0.00090...
- 48 14:15:59 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'no' AND personal_loan = 'yes'; 0.0026 sec / 0.00090...
- 49 14:16:12 ALTER TABLE `Horizon_Bank`.`bank_campaigns` ALTER INDEX `index_personal` INVISIBLE; OK 0.0003 sec
- 50 14:16:14 ALTER TABLE `Horizon_Bank`.`bank_campaigns` ALTER INDEX `index_housing` VISIBLE; OK 0.0003 sec
- 51 14:16:28 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'yes' AND personal_loan = 'no'; 0.0035 sec / 0.00090...
- 52 14:16:28 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'yes' AND personal_loan = 'yes'; 0.0032 sec / 0.00090...
- 53 14:16:27 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'no' AND personal_loan = 'no'; 0.0025 sec / 0.00090...
- 54 14:16:28 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'no' AND personal_loan = 'yes'; 0.0026 sec / 0.00090...
- 55 14:16:34 ALTER TABLE `Horizon_Bank`.`bank_campaigns` ALTER INDEX `index_housing` INVISIBLE; OK 0.0003 sec
- 56 14:16:37 ALTER TABLE `Horizon_Bank`.`bank_campaigns` ALTER INDEX `index_personal` INVISIBLE; OK 0.0003 sec
- 57 14:16:44 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'yes' AND personal_loan = 'no'; 0.0038 sec / 0.0009...
- 58 14:16:44 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'yes' AND personal_loan = 'yes'; 0.0036 sec / 0.0009...
- 59 14:16:45 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'no' AND personal_loan = 'no'; 0.0033 sec / 0.00090...
- 60 14:16:46 SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank_campaigns` WHERE housing_loan = 'no' AND personal_loan = 'yes'; 0.0034 sec / 0.00091...

The queries with index have shown the following results:

- The speed of the results of the query with `index_personal` has an average of 0.0034 sec. (30% efficiency degradation in speed).
- The speed of the results of the query with `index_housing` has an average of 0.0029 sec. (12% efficiency degradation in speed).
- The speed of the results of the query with `index_personal_housing` has an average of 0.0035 sec. (34% efficiency degradation in speed).

3.b Speed Comparison for a complex query on a joint of larger tables (with and without index)

To test if the results of the queries using the index are more efficient (fast) in a larger table and differ from results of a small table, we have included a larger table with 152,000 rows (table

'256X').

The screenshot shows the MySQL Workbench interface with the following details:

- Schemas:** The left sidebar shows the database structure with the '256x' schema selected.
- Query Editor:** The main window displays a complex query involving multiple joins and index creation. The query includes:


```

53   FROM P_Status
54   LEFT JOIN Horizon_Bank.customer USING(Customer_ID)
55   WHERE Priority_Order IN ('High_Priority');

56
57   #write query or use "alter table" to create new index
58   #one-column index
59   CREATE INDEX index_personal ON 256x(personal_loan(10));
60   CREATE INDEX index_housing ON 256x(housing_loan(10));
61   #two-column index
62   CREATE INDEX index_personal_housing ON 256x(personal_loan(10),housing_loan(10));
63
64
65
      
```
- Action Output:** Below the query editor, the action output shows the execution log with the following entries:

Time	Action	Response	Duration / Fetch Time
14:16:09	Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0002 sec / 0.0000...
48	ALTER TABLE `Horizon_Bank`.`bank_campaign` ALTER INDEX `index_personal` INVISIBLE	OK	0.0026 sec / 0.0000...
49	ALTER TABLE `Horizon_Bank`.`bank_campaign` ALTER INDEX `index_housing` INVISIBLE	OK	0.000 sec
50	ALTER TABLE `Horizon_Bank`.`bank_campaign` ALTER INDEX `index_personal_housing` INVISIBLE	OK	0.000 sec
51	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0035 sec / 0.0000...
52	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0033 sec / 0.0000...
53	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0025 sec / 0.0000...
54	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0026 sec / 0.0000...
55	ALTER TABLE `Horizon_Bank`.`bank_campaign` ALTER INDEX `index_housing` INVISIBLE	OK	0.000 sec
56	ALTER TABLE `Horizon_Bank`.`bank_campaign` ALTER INDEX `index_personal_housing` INVISIBLE	OK	0.000 sec
57	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0038 sec / 0.0000...
58	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0036 sec / 0.0000...
59	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0033 sec / 0.0000...
60	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0034 sec / 0.0000...
61	CREATE INDEX index_personal ON 256x(personal_loan(10))	0 row(s) affected Rec... 0.261 sec	
62	CREATE INDEX index_housing ON 256x(housing_loan(10))	0 row(s) affected Rec... 0.136 sec	
63	CREATE INDEX index_personal_housing ON 256x(personal_loan(10),housing_loan(10))	0 row(s) affected Rec... 0.168 sec	

Query without Index:

The screenshot shows the MySQL Workbench interface with the following details:

- Schemas:** The left sidebar shows the database structure with the '256x' schema selected.
- Query Editor:** The main window displays a query to group people by their loan type. The query includes:


```

66
67   Your query example
68   group people by their loan type
69   SELECT
70     Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans
71   FROM
72     256x
73   GROUP BY Housing_Loan , Personal_Loan;
      
```
- Result Grid:** The results show the count of loans grouped by housing and personal loans.

	Housing_Loan	Personal_Loan	Count_of_Loans
no	no	79618	
no	yes	55640	
yes	yes	13624	
- Action Output:** Below the query editor, the action output shows the execution log with the following entries:

Time	Action	Response	Duration / Fetch Time
14:16:44	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0038 sec / 0.0000...
58	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0036 sec / 0.0000...
59	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0033 sec / 0.0000...
60	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `bank`...	4 row(s) returned	0.0034 sec / 0.0000...
61	CREATE INDEX index_personal ON 256x(personal_loan(10))	0 row(s) affected Rec... 0.261 sec	
62	CREATE INDEX index_housing ON 256x(housing_loan(10))	0 row(s) affected Rec... 0.136 sec	
63	CREATE INDEX index_personal_housing ON 256x(personal_loan(10),housing_loan(10))	0 row(s) affected Rec... 0.168 sec	
64	ALTER TABLE `Horizon_Bank`.`256x` ALTER INDEX `index_personal` INVISIBLE	OK	0.050 sec
65	ALTER TABLE `Horizon_Bank`.`256x` ALTER INDEX `index_housing` INVISIBLE	OK	0.000 sec
66	ALTER TABLE `Horizon_Bank`.`256x` ALTER INDEX `index_personal_housing` INVISIBLE	OK	0.000 sec
67	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `256x`...	4 row(s) returned	0.169 sec / 0.000015...
68	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `256x`...	4 row(s) returned	0.160 sec / 0.000016...
69	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `256x`...	4 row(s) returned	0.134 sec / 0.000016...
70	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `256x`...	4 row(s) returned	0.141 sec / 0.000016...
71	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `256x`...	4 row(s) returned	0.139 sec / 0.000013...
72	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `256x`...	4 row(s) returned	0.139 sec / 0.000013...
73	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `256x`...	4 row(s) returned	0.161 sec / 0.000016...
74	SELECT Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans FROM `256x`...	4 row(s) returned	0.164 sec / 0.000013...

The speed of the results of the query without index has an average of 0.140 sec.

Query with Index:

The screenshot shows the MySQL Workbench interface. On the left, the 'Schemas' tree view is open, showing various tables like '256x', '256xx', 'bank_campaigns', and 'contacts'. In the center, a query editor window displays the following SQL code:

```

66
67 Your query example
68 group people by their loan type
69 •
70     SELECT
71         Housing_Loan, Personal_Loan, COUNT(*) AS Count_of_Loans
72     FROM
73         256x
74     GROUP BY Housing_Loan , Personal_Loan;
    
```

Below the query editor is a 'Result Grid' showing the output:

	Housing_Loan	Personal_Loan	Count_of_Loans
Yes	no		79618
No	no		85940
Yes	yes		13824

At the bottom, the 'Action Output' pane shows the execution log with 92 rows, indicating the creation and modification of indexes for the 'Housing_Loan' table. The log includes entries for creating indexes (e.g., 'ALTER INDEX "index_personal"' VISIBLE), selecting data, and dropping indexes (e.g., 'ALTER TABLE "Housing_Loan" - 256x' ALTER INDEX "index_personal_housing" INVISIBLE).

- The speed of the results of the query with index_personal has an average of 0.141 sec. (.07% efficiency degradation of speed).
- The speed of the results of the query with index_housing has an average of 0.140sec. (No changes in speed).
- The speed of the results of the query with index_personal_housing has an average of 0.138 sec. (1.42% efficiency improvement of speed).

Results:

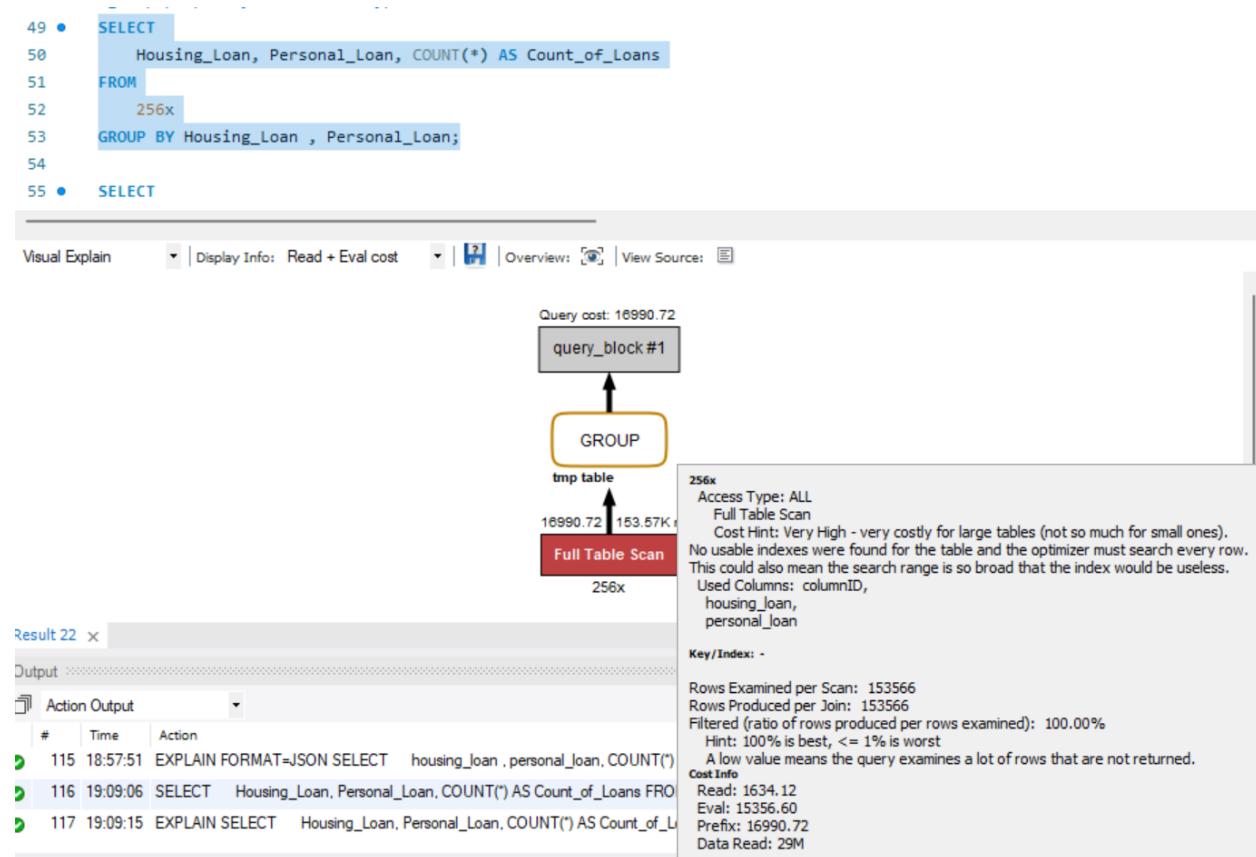
Effectiveness of Indexing: The indexes seem to have a more pronounced effect in larger tables. While the differences in execution times are relatively small in the small table sample, they

become more noticeable in the larger table.

One-Column vs. Two-Column Indexes: The two-column composite index (index_personal_housing) appears to provide a slight performance benefit in the larger table. This suggests that composite indexes can be particularly effective for queries that involve multiple fields.

Prefix Indexing: Using prefix indexes might have impacted the performance differently depending on the distribution of data in the personal_loan and housing_loan columns. Prefix indexing is a trade-off between index size and query performance.

4. Explainer Diagram



In the first screenshot, The query has a high cost, indicating its inefficiency. It is performing a full table scan, meaning it is examining every row(153.58k rows) in the table rather than using

indexes to narrow down the search, which is typically slower.

```

55 • SELECT
56   housing_loan , personal_loan, COUNT(*) AS Count_of_Loans
57   FROM
58     256x
59   WHERE
60     housing_loan = 'yes'
61   GROUP BY housing_loan , personal_loan;
62
63 • SELECT

```

Visual Explain | Display Info: Read + Eval cost | ? | Overview: | View Source: |

Result Grid | Form Editor | Field Types | Query Stats

query_block #1 (Query cost: 12580.66)

GROUP

tmp table (Non-Unique Key Lookup, 12580.66, 76.78K rows)

Attached Condition: ('horizon_bank'.`256x`.`housing_loan` = 'yes')

Action Output

#	Time	Action
113	18:57:45	SELECT housing_loan , personal_loan, COUNT(*) AS Count_of_Loans FROM 256x
114	18:57:51	EXPLAIN SELECT housing_loan , personal_loan, COUNT(*) AS Count_of_Loans FR
115	18:57:51	EXPLAIN FORMAT=JSON SELECT housing_loan , personal_loan, COUNT(*) AS Count_of_Loans

Rows Examined per Scan: 76783
Rows Produced per Join: 76783
Filtered (ratio of rows produced per rows examined): 100.00%
Hint: 100% is best, <= 1% is worst
A low value means the query examines a lot of rows that are not returned.
Cost Info:
Read: 4902.36
Eval: 7678.30
Prefix: 12580.66
Data Read: 14M

In the second screenshot, the clause is modified by adding a where function to activate the index. The index being used is index_housing, which is based on the housing_loan column in the original dataset. In this result, The number of rows examined per scan has been decreased to 86.7k rows, the efficiency has been raised successfully, but needs further improvement.

55 • SELECT
56 housing_loan , personal_loan, COUNT(*) AS Count_of_Loans
57 FROM
58 256x
59 WHERE
60 housing_loan = 'yes' AND personal_loan = 'yes'
61 GROUP BY housing_loan , personal_loan;

Visual Explain | Display Info: Read + Eval cost | Overview: | View Source:

256x
Access Type: ref
Non-Unique Key Lookup
Cost Hint: Low-medium - Low if number of matching rows is small, higher as the number of rows increases.
Used Columns: housing_loan,
personal_loan

Key/Index: index_personal_housing
Ref.: const,
const
Used Key Parts: personal_loan,
housing_loan
Possible Keys: index_personal,
index_housing,
index_personal_housing

Attached Condition:
('horizon_bank'.`256x`.personal_loan` = 'yes')
AND ('horizon_bank'.`256x`.housing_loan` = 'yes')

Rows Examined per Scan: 27200
Rows Produced per Join: 27200
Filtered (ratio of rows produced per rows examined): 100.00%
Hint: 100% is best, <= 1% is worst
A low value means the query examines a lot of rows that are not returned.

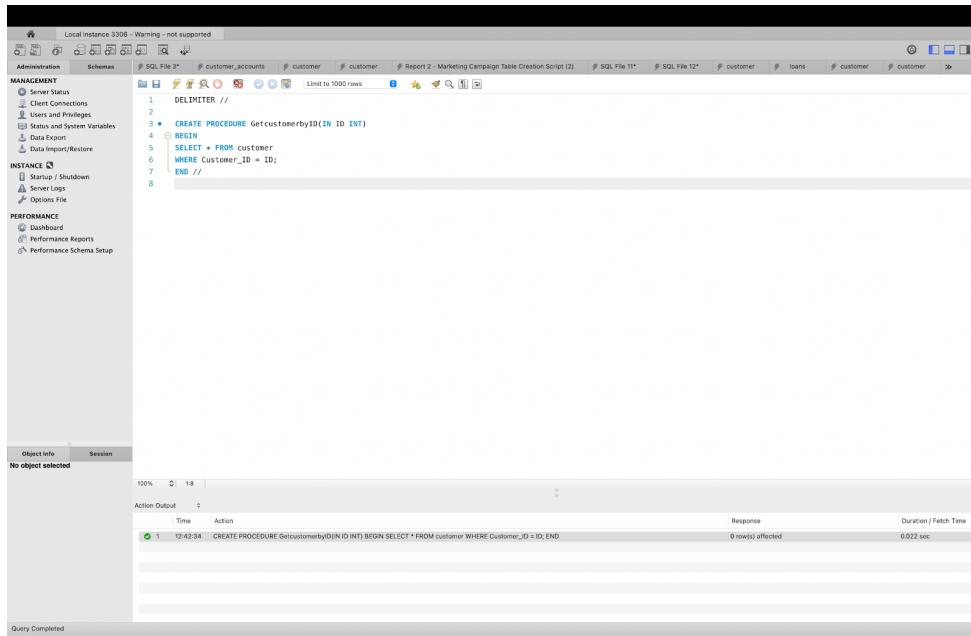
Cost Info:
Read: 4902.36
Eval: 2720.00
Prefix: 7622.36
Data Read: 9M

Result 15 x
Output
Action Output
Time Action
94 18:00:42 ALTER TABLE `horizon_bank`.`256x` ALTER INDEX `index_personal_housing` VISIBLE
95 18:00:51 SELECT housing_loan , personal_loan, COUNT(*) AS Count_of_Loans FROM 256x
96 18:00:58 EXPLAIN SELECT housing_loan , personal_loan, COUNT(*) AS Count_of_Loans FROM 256x

In the third screenshot, by modifying the where clause again, this query can use the two-column index created before successfully. The query utilizes a composite index :index_personal_housing. This two-column index allows the query to reference these two fields quickly, significantly reducing the number of rows that need to be examined. However, since it still involves checking 27.2k rows(the original one is 153.57k rows, and the one-column index is 86.7k rows), it suggests that the index may still not be optimal, or the query conditions may not entirely align with the best use scenario for the index.

5. Stored Procedures: three (3) different Stored Procedures using different parameters.

Stored Procedure 1: Get customer by ID



The screenshot shows the SQL Server Management Studio interface. In the center, there is a code editor window titled "SQL File 3" containing the following T-SQL code:

```
1  DELIMITER //  
2  //  
3  CREATE PROCEDURE GetcustomerbyID(IN ID INT)  
4  BEGIN  
5      SELECT * FROM customer  
6      WHERE Customer_ID = ID;  
7  END //
```

Below the code editor, the "Session" tab is selected in the bottom-left corner. Under the "Session" tab, there is a table titled "Action Output" showing the execution details of the stored procedure creation:

Action	Time	Action	Response	Duration / Fetch Time
CREATE PROCEDURE GetcustomerbyID(IN ID INT) BEGIN SELECT * FROM customer WHERE Customer_ID = ID; END	12:42:34		0 rows(s) affected	0.022 sec

At the bottom of the interface, a message "Query Completed" is displayed.

The SQL code above defines a stored procedure named ‘GetcustomerbyID’ that retrieves customer information from the “customer” table based on the specific customer ‘ID’. The procedure takes an input parameter ID, and when called with a customer value, in this case 505721, using ‘CALL GetcustomerID(505721)’, it executes the SELECT query, fetching all columns for the customer matching Customer_ID (505721).

Result:

```

1 • CALL GetcustomerbyID(600721)

```

Customer_ID	MaritalStatus	First_Name	Last_Name	Nationality	Age	Job	Education
600721	Married	Svetlana	Koreyshik	Gray	67	retired	other

Action Output

Time	Action	Response	Duration / Fetch Time
12:43:54	CREATE PROCEDURE GetcustomerbyID(N ID INT) BEGIN SELECT * FROM customer WHERE Customer_ID = ID; END	0 rows(affected)	0.022 sec
12:43:55	CALL GetcustomerbyID(600721)	1 row(s) returned	0.048 sec / 0.00045...

Stored Procedure 2: Get customer by Status

```

1 DELIMITER //
2
3 • CREATE PROCEDURE Getcustomerbystatus(IN status TEXT)
4   BEGIN
5     SELECT * FROM customer_accounts
6     JOIN customer USING(Customer_ID)
7     WHERE Balance >= 5200
8     AND MaritalStatus = status
9     ORDER BY Balance DESC;
10   END// |

```

Action Output

Time	Action	Response	Duration / Fetch Time
12:46:11	CREATE PROCEDURE Getcustomerbystatus(N ID INT) BEGIN SELECT * FROM customer WHERE Customer_ID = ID; END	0 rows(affected)	0.019 sec
12:46:22	CALL Getcustomerbystatus(600721)	1 row(s) returned	0.0042 sec / 0.00001...
12:46:30	CREATE PROCEDURE Getcustomerbystatus(IN status TEXT) BEGIN SELECT * FROM customer_accounts LEFT JOIN customer USING(Customer_ID) WHERE Balance >= ...;	0 rows(affected)	0.0052 sec

The SQL code above defines a stored procedure named ‘Getcustomerbystatus’ that retrieves customer information from the “customer accounts” table, joined with the “customer” table, based on a specified marital status and a minimum balance requirement. The procedure takes an input parameter “status”, and when called, it executes the SELECT query, fetching all columns for customers with a balance greater than or equal to 5200 and matching the provided marital

status. The result is ordered by balance in descending order. The stored procedure is then called with the argument “Married” to fetch relevant customer information.

Result:

```

Local Instance 3306 - Warning - not supported
Management Schemas SQL File 1* # customer_accounts # customer # customer # Report 2 - Marketing Campaign Table Creation Script (2) SQL File 11* SQL File 12* # loans # customer # customer >
ADMINISTRATION
    ⚡ Server Status
    ⚡ Client Connections
    ⚡ Users and Privileges
    ⚡ Status and System Variables
    ⚡ Data Import/Export
    ⚡ Data Import/Restore
INSTANCE
    ⚡ Startup / Shutdown
    ⚡ Server Logs
    ⚡ Options File
PERFORMANCE
    ⚡ Dashboard
    ⚡ Performance Reports
    ⚡ Performance Schema Setup
Object Info Session No object selected
Result 2
100% 36/1
Result Grid Filter Rows Export: 
Customer_ID Balance Marital_Status First_Name Last_Name Nationality Age Job Education
685090 12411 Married Hengri Irshen US 38 manager tertiary
685618 16957 Married Cason Lawson US 38 manager tertiary
685620 16958 Married Kaitlin Kaitlin US 39 manager tertiary
593388 15458 Married Mokkena Koetz US 29 management tertiary
684970 12907 Married Salvatore Myrna US 39 management tertiary
684433 12569 Married Corinna Ramsey US 31 management tertiary
684434 11982 Married Boleslaw Boleslaw US 39 manager secondary
711188 8876 Married Bellamy Wall US 38 services secondary
684435 12568 Married Christopher Powell US 33 business support
694163 8556 Married Brodie Jackson US 79 retired primary
694164 8557 Married Aileen Calvert US 47 retired secondary
631771 8436 Married Douglas Douglas US 32 management secondary
695890 8304 Married Alan Church US 60 retired secondary
695891 8305 Married Franklin Franklin US 39 services secondary
732893 8234 Married Teagan Durham US 34 management tertiary
684436 8235 Married Blair Tabor US 33 technician tertiary
549407 8236 Married Zeriyah Woodland US 33 technician tertiary
710074 7792 Married Penny Hodges US 41 blue-collar tertiary
605345 7549 Married Maria Lozano US 35 technician tertiary
605346 7550 Married Dale Harrington US 35 technician secondary
583326 7108 Married Elsie Zamora US 45 technician other
674409 6882 Married Alton Collier US 29 technician secondary
782727 6883 Married Sarah Burns US 70 retired secondary
674409 6738 Married Hayden Reyna US 38 technician secondary
674409 6694 Married Christopher Bowen US 35 blue-collar secondary
574807 6411 Married Bowen Aguilar US 35 blue-collar secondary
574808 6402 Married Meagan Kent US 38 technician secondary

```

Action Output:

Time	Action	Response	Duration / Fetch Time
12:46:11	CREATE PROCEDURE GetCustomerByStatus(IN ID INT) BEGIN SELECT * FROM customer WHERE Customer_ID = ID; END	0 rows affected	0.013 sec
12:46:22	CALL GetCustomerByStatus(50572)	1 row(s) returned	0.0042 sec / 0.0001...
4	CREATE PROCEDURE GetCustomerByStatus(IN status TEXT) BEGIN SELECT * FROM customer_accounts LEFT JOIN customer USING(Customer_ID) WHERE Balance >= 5...	0 rows affected	0.0009 sec
5	CALL GetCustomerByStatus('Married')	41 row(s) returned	0.069 sec / 0.00044...
6	CREATE PROCEDURE GetCustomerByJobTitle(IN status TEXT) BEGIN SELECT * FROM customer_accounts LEFT JOIN customer USING(Customer_ID) WHERE Job = status...	0 rows affected	0.053 sec

Query Completed

Stored Procedure 3: Get customer by Job Title

```

Local Instance 3306 - Warning - not supported
Management Schemas SQL File 1* # customer_accounts # customer # customer # Report 2 - Marketing Campaign Table Creation Script (2) SQL File 11* SQL File 12* # loans # customer # customer >
ADMINISTRATION
    ⚡ Server Status
    ⚡ Client Connections
    ⚡ Users and Privileges
    ⚡ Status and System Variables
    ⚡ Data Import/Export
    ⚡ Data Import/Restore
INSTANCE
    ⚡ Startup / Shutdown
    ⚡ Server Logs
    ⚡ Options File
PERFORMANCE
    ⚡ Dashboard
    ⚡ Performance Reports
    ⚡ Performance Schema Setup
Object Info Session No object selected
Result 2
100% 6/9
Action Output:
1  DELIMITER //
2
3  CREATE PROCEDURE GetCustomerByJobTitle(IN status TEXT)
4      BEGIN
5          SELECT * FROM customer_accounts
6              LEFT JOIN customer USING(Customer_ID)
7          WHERE Job = status
8          ORDER BY Balance DESC;
9      END//|

```

Action Output:

Time	Action	Response	Duration / Fetch Time
12:46:11	CREATE PROCEDURE GetCustomerByJobTitle(IN ID INT) BEGIN SELECT * FROM customer WHERE Customer_ID = ID; END	0 rows affected	0.013 sec
12:46:22	CALL GetCustomerByJobTitle(50572)	1 row(s) returned	0.0042 sec / 0.0001...
4	CREATE PROCEDURE GetCustomerByJobTitle(IN status TEXT) BEGIN SELECT * FROM customer_accounts LEFT JOIN customer USING(Customer_ID) WHERE Balance >= 5...	0 rows affected	0.0009 sec
5	CALL GetCustomerByJobTitle('Married')	41 row(s) returned	0.069 sec / 0.00044...
6	CREATE PROCEDURE GetCustomerByJobTitle(IN status TEXT) BEGIN SELECT * FROM customer_accounts LEFT JOIN customer USING(Customer_ID) WHERE Job = status...	0 rows affected	0.053 sec

Query Completed

The SQL code above creates a stored procedure names “Getcustomerbyjobtitle” that retrieves customer information from the “customer_accounts” table, joined with the “customer” table. It filters the result based on a specific job title (“management”) and orders it by balance in descending order. The stored procedure is then called the argument ‘management’ to fetch relevant customer information.

Result:

The screenshot shows the SQL Server Management Studio interface. The left sidebar displays the Object Explorer with nodes for MANAGEMENT, INSTANCE, and PERFORMANCE. The main pane shows the results of a query:

```

1 • CALL Getcustomerbyjobtitle('management')

```

The results grid displays customer information:

Customer_ID	Balance	MaritalStatus	First_Name	Last_Name	Nationality	Age	Job	Education
651388	15469	Married	Carmen	Koch	US	29	management	tertiary
548870	12807	Married	Salvatore	Myers	US	32	management	tertiary
631771	12808	Married	Christopher	Riley	US	31	management	tertiary
773797	6848	Single	Douglas	Michael	US	32	management	tertiary
651389	8530	Divorced	Rebecca	Allen	US	34	management	tertiary
610358	4761	Married	Everest	Aguilar	US	32	management	tertiary
630028	4415	Married	Jessica	Ellison	US	48	management	secondary
780062	3884	Divorced	River	Wren	US	37	management	primary
602110	3691	Single	Kimberly	Branco	US	33	management	tertiary
716871	3676	Married	Troy	Glover	US	38	management	other
642023	3321	Married	Grace	Anneweid	US	34	management	tertiary
549055	2964	Married	Isaac	Foster	US	37	management	tertiary
531078	2587	Married	Eli	Sisko	US	55	management	primary
684843	2026	Married	Lena	Hoskell	US	55	management	tertiary
588145	1831	Married	Kai	Potter	US	48	management	tertiary
589918	1944	Married	Jane	Georgina	US	52	management	tertiary
634881	1727	Married	Don	Selena	US	59	management	primary
498116	1696	Married	Barbara	Stanton	US	43	management	tertiary

The bottom pane shows the Action Output log:

Action	Time	Action	Response	Duration / Fetch Time
1	12:48:11	CREATE PROCEDURE Getcustomerbyjobtitle(@ID INT) BEGIN SELECT * FROM customer WHERE Customer_ID = @ID; END	0 row(s) affected	0.013 sec
2	12:48:22	CALL Getcustomerbyjobtitle(1)	1 row(s) returned	0.0042 sec / 0.00001...
3	12:48:22	CREATE PROCEDURE Getcustomerbystatus(@IN status TEXT) BEGIN SELECT * FROM customer_accounts LEFT JOIN customer USING(Customer_ID) WHERE Balance >= @IN; END	0 row(s) affected	0.0053 sec
4	12:49:10	CALL Getcustomerbystatus('Married')	41 row(s) returned	0.009 sec / 0.00044...
5	12:49:02	CREATE PROCEDURE Getcustomerbyjobtitle(@IN status TEXT) BEGIN SELECT * FROM customer_accounts LEFT JOIN customer USING(Customer_ID) WHERE Job = @IN; END	0 row(s) affected	0.053 sec
6	12:49:46	CALL Getcustomerbyjobtitle('management')	131 row(s) returned	0.056 sec / 0.00061...

6. Triggers: three (3) different types of triggers: insert, update, and delete.

Trigger1: After Insert

```

64      #trigger: pick those people who may need another telephone campaign
65      #(those who first telephone duration is under 10s)
66 • CREATE TABLE recall (
67      recall_id INT AUTO_INCREMENT PRIMARY KEY,
68      action_type VARCHAR(50),
69      duration_second INT,
70      subscribe_result VARCHAR(255),
71      action_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
72      whether_recall VARCHAR(255)
73 );
74
75  DELIMITER //
76
77 • CREATE TRIGGER 256x_AFTER_INSERT
78  AFTER INSERT ON `256x`
79  FOR EACH ROW
80  BEGIN
81  •     IF NEW.duration_second < 10 THEN
82      INSERT INTO recall (recall_id, duration_second, subscribe_result, whether_recall)
83      VALUES (NEW.columnID, NEW.duration_second, NEW.subscribe_result, 'yes');
84  END IF;
85 END;
86 //
87
88  DELIMITER ;
89
90 • #insert
91  INSERT INTO 256x (columnID, duration_second, subscribe_result)
92  VALUES ('250080','7', 'no');
93
94 • INSERT INTO 256x (columnID, duration_second, subscribe_result)
95  VALUES ('289023','5', 'no');
96
97 • INSERT INTO 256x (columnID, duration_second, subscribe_result)
98  VALUES ('289899','9', 'no');

```

Query Description:

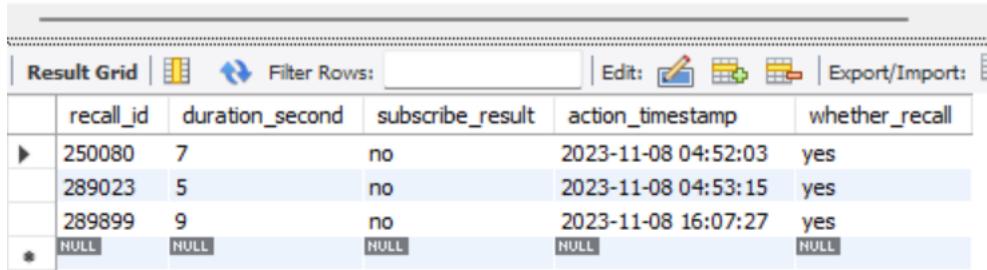
This query is designed to identify customers whose initial telephone campaign duration is less than ten seconds and categorize them as 'people in need of a second call.' They will be added to a new table named 'recall.'

First, I created the 'recall' table. Then, I establish an after-insert trigger. This trigger activates

when new rows are inserted into the '256x' table (which contains comprehensive customer information). If the call duration in the newly inserted rows is under 10 seconds, three pieces of personal information, including 'columnID,' 'duration,' and 'subscribe result,' will also be inserted into the 'recall' table. Finally, I insert three sample records to test whether this trigger runs correctly.

Result:

```
1 •  SELECT * FROM horizon_bank.recall;
```



The screenshot shows a MySQL Workbench result grid with the following data:

	recall_id	duration_second	subscribe_result	action_timestamp	whether_recall
▶	250080	7	no	2023-11-08 04:52:03	yes
	289023	5	no	2023-11-08 04:53:15	yes
*	289899	9	no	2023-11-08 16:07:27	yes
*	HULL	HULL	HULL	NULL	NULL

Trigger 2: Before Update

```
101  # trigger 2: find out those people who may need a third call and record them.  
102  #(second_durataion > 10s and subscribe_result_2 = 'no')  
103  
104  #add a column 'recall_situation' to recall table  
105 • ALTER TABLE recall  
106    ADD second_duration INT,  
107    ADD subscribe_result_2 VARCHAR(10),  
108    ADD whether_third_recall VARCHAR(10);  
109  
110  DELIMITER //  
111 • CREATE TRIGGER BEFORE_RECALL_UPDATE BEFORE UPDATE ON recall  
FOR EACH ROW  
113  BEGIN  
114  IF NEW.second_duration < 10 THEN  
115    SET NEW.whether_third_recall = 'no';  
116  ELSEIF NEW.second_duration >= 10 AND NEW.subscribe_result_2 = 'no' THEN  
117    SET NEW.whether_third_recall = 'yes';  
118  ELSEIF NEW.subscribe_result_2 = 'yes' THEN  
119    SET NEW.whether_third_recall = 'no';  
120  END IF;  
121  END;  
122  //  
123  DELIMITER ;
```

```

125 •  # update recall table
126   UPDATE recall
127   SET
128   second_duration = 15 ,
129   subscribe_result_2 = 'no'
130   WHERE
131   recall_id = 250080;
132
133 •  UPDATE recall
134   SET
135   second_duration = 3,
136   subscribe_result_2 = 'no'
137   WHERE
138   recall_id = 289023;
139
140 •  UPDATE recall
141   SET
142   second_duration = 121,
143   subscribe_result_2 = 'yes'
144   WHERE
145   recall_id = 289899;

```

Query Description:

This query is designed to identify customers whose second telephone campaign duration is over 10 seconds but still not subscribe for the product. They will be automatically marked as ‘people need a third call’, and the others should be marked as ‘people do not need a third call’ in the ‘recall’ table.

First, I created three new columns for the recall table to record the result of the second telephone campaign. Then, I establish a before-update trigger. This trigger is activated when some information is inserted into those three new columns. When the second telephone campaign duration(second_duration) is over 10 seconds and the second subscription result(subscribe_result_2) is still ‘no’, the ‘whether_third_recall’ will be input by ‘yes’ automatically. Finally, I update three sample records to test whether this trigger runs correctly.

Result:

Result Grid								
	recall_id	duration_second	subscribe_result	action_timestamp	whether_recall	second_duration	subscribe_result_2	whether_third_recall
▶	250080	7	no	2023-11-08 04:52:03	yes	15	no	yes
	289023	5	no	2023-11-08 04:53:15	yes	3	no	no
*	289899	9	no	2023-11-08 16:07:27	yes	121	yes	no
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Trigger 3: Before Delete

```
148      #trigger 3: delete and record those unpotential customers of this product
149      # those who subscribe_result_2 and whether_third_recall are both 'no'
150
151 • CREATE TABLE unpotential_customer(
152     recall_id INT AUTO_INCREMENT PRIMARY KEY
153 );
154
155     DELIMITER //
156 •  CREATE TRIGGER before_recall_delete
157     BEFORE DELETE ON recall
158     FOR EACH ROW
159     BEGIN
160         IF OLD.subscribe_result_2 = 'no' AND OLD.whether_third_recall = 'no' THEN
161             INSERT INTO unpotential_customer SET recall_id = OLD.recall_id;
162         END IF;
163     END;
164     //
165     DELIMITER ;
166
167 •  #delete
168     DELETE FROM recall
169     WHERE subscribe_result_2 = 'no' AND whether_third_recall = 'no'
170     LIMIT 10;
```

Query Description:

This query is designed to identify those people who do not subscribe for the

product and also do not need a third call. Those customers will be marked as 'potential customers'.

Firstly, I created the 'unpotential_customer' table. Then, a before delete trigger is defined, which is programmed to act before a deletion operation is executed on the recall table. It examines the row which meets specific criteria: both the subscribe_result_2 and whether_third_recall fields are set to 'no'. If the row meets these conditions, it triggers an action that inserts the recall_id from this row into an unpotential_customer table. Finally, I run a sample record to test whether this trigger runs correctly.

Result:

```
1 •  SELECT * FROM horizon_bank.recall;
2 •  SELECT * FROM unpotential_customer;
```

Result Grid								
	recall_id	duration_second	subscribe_result	action_timestamp	whether_recall	second_duration	subscribe_result_2	whether_third_recall
▶	250080	7	no	2023-11-08 04:52:03	yes	15	no	yes
▶	289899	9	no	2023-11-08 16:07:27	yes	121	yes	no
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
1 •  SELECT * FROM horizon_bank.recall;
2 •  SELECT * FROM unpotential_customer;
```

Result Grid	
	recall_id
▶	289023
*	NULL