

**Database Project: Customer Data for Targeting in the Telebanking**

ISM 6218: Advanced Database Management

Professor Don Berdnt

April 26, 2024

Gang Wei, Zahid Rahman,Asif Imtiaz, Long Nguyen

A grey logo on a black background

Description automatically generated

Table of Contents

[Executive Summary 2](#_Toc165067665)

[Part 1: Database Design 4](#_Toc165067666)

[1.1. Conceptual Design 4](#_Toc165067667)

[1.2. Entity-Relationship Diagram (ERD) 6](#_Toc165067668)

[Part 2: Query Writing 8](#_Toc165067669)

[2.1. Point Query 8](#_Toc165067670)

[2.2. Range Query 9](#_Toc165067671)

[2.3. Scan Query 9](#_Toc165067672)

[Part 3: Performance Tuning 10](#_Toc165067673)

[3.1. Partitioning 10](#_Toc165067674)

[3.2. Indexing 12](#_Toc165067675)

[a. Point query with indexed column 12](#_Toc165067676)

[b. Range query with indexed column 13](#_Toc165067677)

[c. Scan query with indexed column 13](#_Toc165067678)

[3.3. Parallelism 13](#_Toc165067679)

[Part 4: Additional Topic 14](#_Toc165067680)

[4.1. Data Visualization 14](#_Toc165067681)

[4.2. Detecting Outliers – Data Mining 16](#_Toc165067682)

[4.3. DBA scripts 18](#_Toc165067683)

# Executive Summary

As we know, one of the most important profit points for the banks, especially commercial banks, would be loaning their customers’ deposits out at a higher rate. Sufficient funds are of great importance for the banks. To target the appropriate customers fast and correctly would gain most of depositors as soon as possible for the competitive advantage. Therefore creating an engaging and rewarding marketing campaign in banking to invite new customers has been a great challenge. Banks need to be precise in their targeting strategy to allocate their resources efficiently for the best gains in a timely manner. Targeting in marketing campaigns correctly requires sufficient data about customers, thus requiring a functioning database system.

With that in mind, our project will focus on developing a comprehensive database for a financial institution, specifically storing the data about their customers. The database will consist of information regarding the customer’s financial background, their current business with the bank, and whether they have subscribed to the bank term deposit offer or not. The data would be useful in two cases: (1) when the bank wants to analyze their customer base and evaluate their marketing success, and (2) to develop machine learning model to help target/predict the right customers for their future campaigns using different attributes.

For the sake of this project, we have used the data from the UCI Machine Learning Repository to distribute our database. The data is a report of direct telemarketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls to a list of clients, accessing if the product (bank term deposit) would be (‘yes’) or not (‘no’) subscribed. The data consists of 45,211 instances with sixteen features, divided into tables.

Below are the group-assigned weights for this project:

|  |  |  |
| --- | --- | --- |
| **Topic Area** | **Description** | **Points** |
| **Database Design** | This part should include a logical database design (for the relational model), using normalization to control redundancy and integrity constraints for data quality. | 30 |
| **Query Writing** | This part is another chance to write SQL queries, explore transactions, and even do some database programming for stored procedures. | 30 |
| **Performance Tuning** | In this section, you can capitalize and extend your prior experiments with indexing, optimizer modes, partitioning, parallel execution, and any other techniques you want to further explore. | 25 |
| **Other Topics** | Here you are free to explore any other topics of interest. Suggestions include DBA scripts, database security, interface design, data visualization, data mining, and NoSQL databases. | 15 |

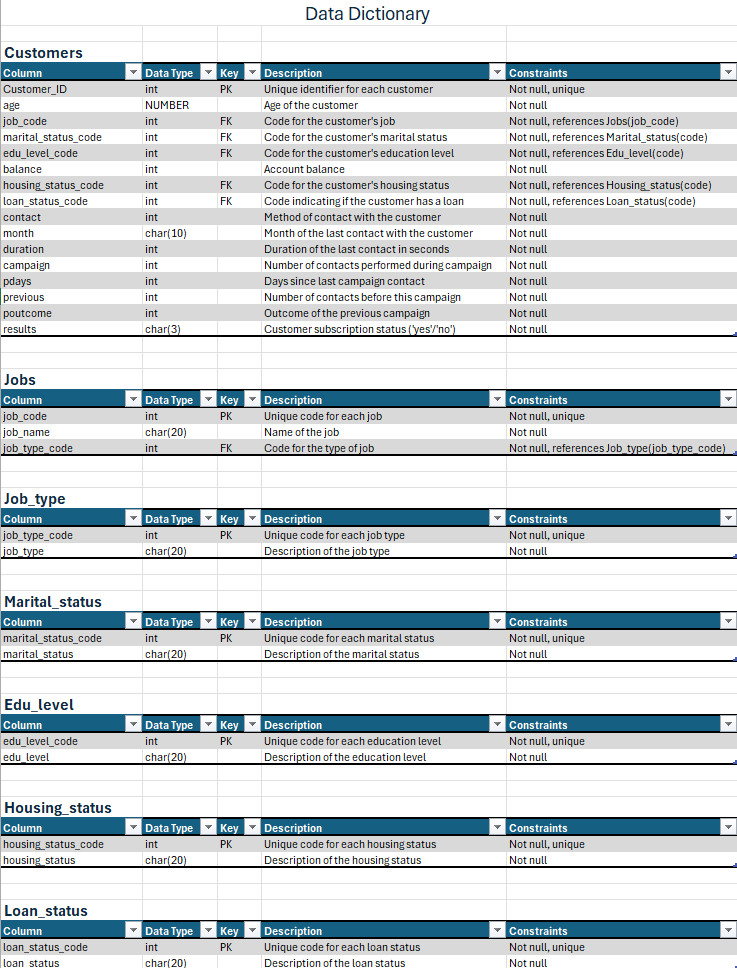
# Part 1: Database Design

## Conceptual Design

A diagram of a company

Description automatically generated

The conceptual design of the entity-relationship diagrams (ERD) above gives us the basic blueprint of the database. Our database has quite a simple design, with CUSTOMERS as the major entity with all the crucial information about the bank’s customers. Other entities help explain the encoded categories in the CUSTOMERS table, such as MARITAL\_STATUS which is categorized as {‘0’: ‘divorced’, ‘1’: ‘single’, ‘2’: ‘married’}. The relationship of CUSTOMERS to every other entity is many-to-one, as each customer has one job, but the same job can be seen among multiple individuals. Realistically, a person can have several jobs, but it is not the case in the bank’s data for customers’ current status. Details about each entity are explained in the data dictionary below:



## Entity-Relationship Diagram (ERD)

A diagram of a computer

Description automatically generated

We used the data definition language (DDL) CREATE TABLE statement to start implementing the CUSTOMERS table. This is our major entity so it will contain all the data necessary about the customers:

-- Create the customers table and set constraints

Create table customers

(

customer\_id int not null primary key,

age number not null,

job\_code int not null,

marital\_status\_code int not null,

edu\_level\_code int not null,

balance int not null,

housing\_status\_code int not null,

loan\_status\_code int not null,

contact int not null,

month char(10) not null,

duration int not null,

campain int not null,

pdays int not null,

previous int not null,

poutcome int not null,

results char(3) not null

)

The data uses a combination of number, integer, and character data types. The table also includes the basic NOT NULL integrity constraints for all the attributes. It is reasonable that the bank collected all the data from their customers necessary for the precise targeting strategy, thus we expected all the columns to have data. We added a primary constraint to the CUSTOMER\_ID column as it is the only candidate key in the table. The primary key constraints will ensure the table is properly structured and normalized, enhance data retrieval for queries and enforce indexing that we created.

-- Create the jobs table and set constraints

Create table jobs  
    (  
    job\_code int not null primary key,  
    job\_name char(20) not null,  
    job\_type\_code int not null  
    );

To keep the report short, we only include the JOBS entity here. However, other tables in the database that are used to clarify the features of the CUSTOMERS entity follow the same structure. In each of these entities, we have the customer’s attributes, such as MARITAL\_STATUS\_CODE or JOB\_CODE in this case, as the primary key. Also, like the CUSTOMER entity, every other column has the NOT NULL integrity constraints to ensure all data is available for future use.

A screenshot of a computer

Description automatically generated

Finally, we wanted to establish relationships between tables, so we added foreign key constraints to the CUSTOMERS for the columns above. This ensures that every foreign key value in the CUSTOMERS entity corresponds to a primary key value in another entity. Adding foreign key constraints helps maintain data quality and reliability in our database.

# Part 2: Query Writing

## 2.1. Point Query

-- Point query (without index)

SELECT max(balance) from customers;

SELECT \*

FROM customers

WHERE balance = ‘102127’;

A screenshot of a computer

Description automatically generated

In part 2, we will begin writing some queries to see if the database functions appropriately, starting with a point query identifying the customer with the highest balance in their account. This query consists of two parts. The first part calculates the maximum BALANCE value from the CUSTOMERS table, which returns ‘102127’. The second part retrieves all columns for customers whose balance is exactly ‘102127’.

To evaluate the query execution, we will focus on the consistent gets and some other metrics such as cost. Consistent gets are the normal block I/O operations using read consistent mode which are typically the most common block reads. There are other measures like DB block gets and physical reads, we focus on the consistent gets from the Autotrace here. Without an index, both queries will require full table scans, resulting in an estimate cost of 426 and consistent gets of 2362, which are slow and not optimal.

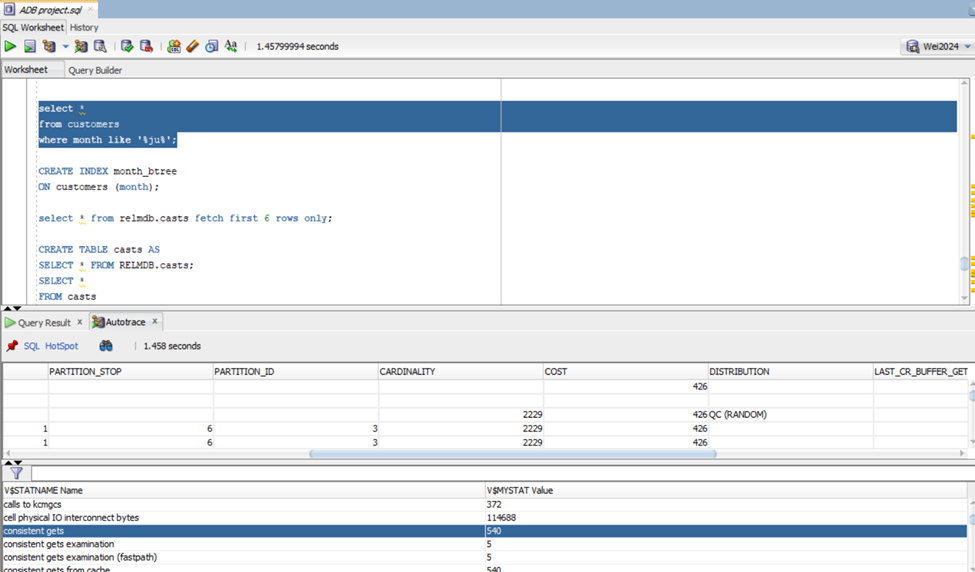
## 2.2. Range Query

-- Range query (without index)

SELECT \*

FROM customers

WHERE month LIKE ‘%ju%’;



The second query we tried to run is a range query, in which we wanted to select all the customers that the bank last contacted in June and July. The idea might be to whom the bank has recently contacted, perhaps to check on their response or incentivize them with a second offer. Since this is an intensive range query over a high number of rows, the query returns cardinality of 2229 and estimate cost of 426. Consistent gets returned is 540, indicating quite a high number of I/O activities during the execution, which can be further improved through indexing.

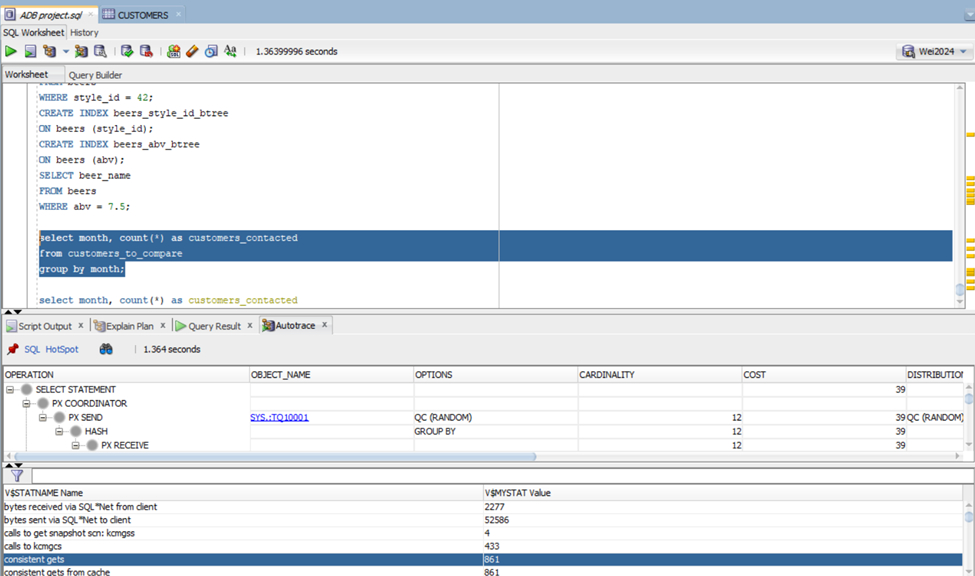
## 2.3. Scan Query

-- Scan query (without index)

SELECT month, COUNT(\*) as customers\_contacted

FROM customers

GROUP BY month;



The last query we ran was a ran query. This query performs a full table scan to aggregate the number of customers contacted per month using COUNT(\*). The GROUP BY MONTH clause then groups the customer data by month. This can provide useful information on how productive the sales team from the bank has been throughout the months of each year. The result can also imply the seasonal/monthly fluctuation in the number of customers called. The Autotrace result shows a also high consistent gets of 861 from this query execution.

# Part 3: Performance Tuning

## 3.1. Partitioning

Partitioning involves dividing large tables or indexes into smaller segments called partitions that can be stored in different file groups. Partitioning uses specific criteria such as range, list, or hash. This method can enhance query performance by enabling parallelism and allowing queries to target specific partitions, reducing the amount of data scanned for operations. Additionally, partitioning can enhance availability and scalability by enabling more efficient data distribution and management across multiple storage devices or servers.

-- Partition the customers table

PARTITION BY RANGE (age)

(

PARTITION p1 VALUES less than (30),

PARTITION p2 VALUES less than (35),

PARTITION p3 VALUES less than (40),

PARTITION p4 VALUES less than (45),

PARTITION p5 VALUES less than (50),

PARTITION p6 VALUES less than (95)

); A screenshot of a computer

Description automatically generated

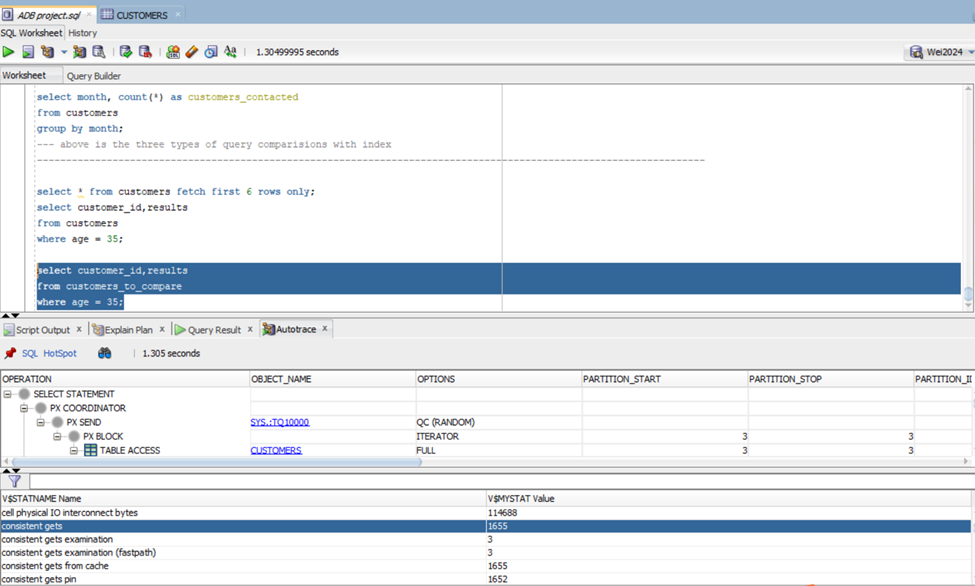
By implementing partitioning, we aim to improve the performance and manageability of the database. The PARTITION BY RANGE (AGE) below was included when we deployed the CREATE TABLE CUSTOMERS code. This technique will divide the CUSTOMERS entity into six smaller partitions specified by their age. We chose range partitioning over list or hash because it is the best fit for the sequential AGE values, as well as the fact that it allows new data entries to be automatically placed into the age-based partition with ease. In order to evaluate the improvement in performance, we ran the query below before and after partitioning.

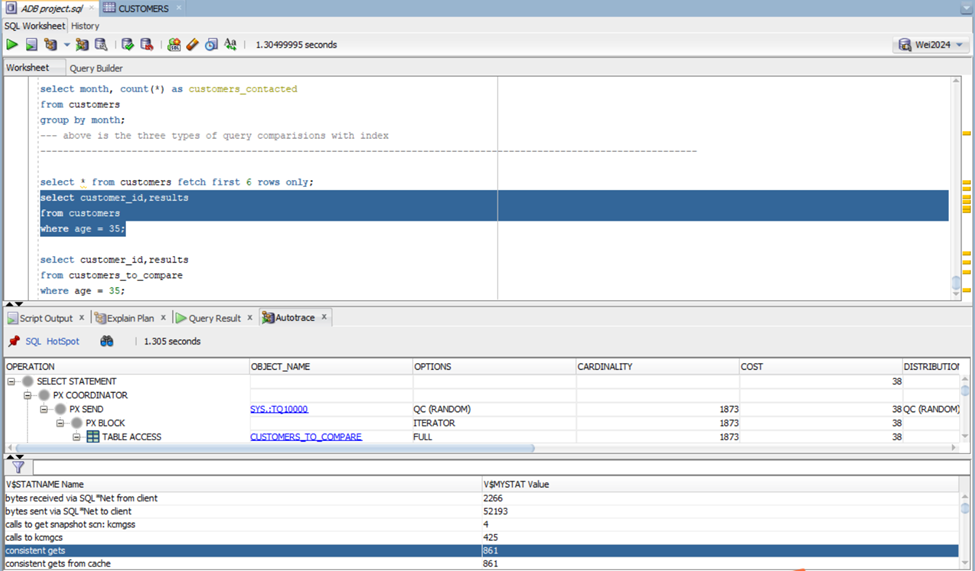
-- Query to test database performance with partitioning

SELECT customer\_id, results

FROM customers

WHERE age = 35;





## 3.2. Indexing

The second performance tuning technique we applied is indexing. Indexing improves database performance by facilitating quick data retrieval and efficient query processing. When we set indexes on specific columns in a database table, we essentially create a data structure that allows the database management system to locate rows quickly using the indexed column values. In our database, we chose MONTH and BALANCE columns to create indexes on:

-- Create indexes on the customer entity

CREATE INDEX month\_btree

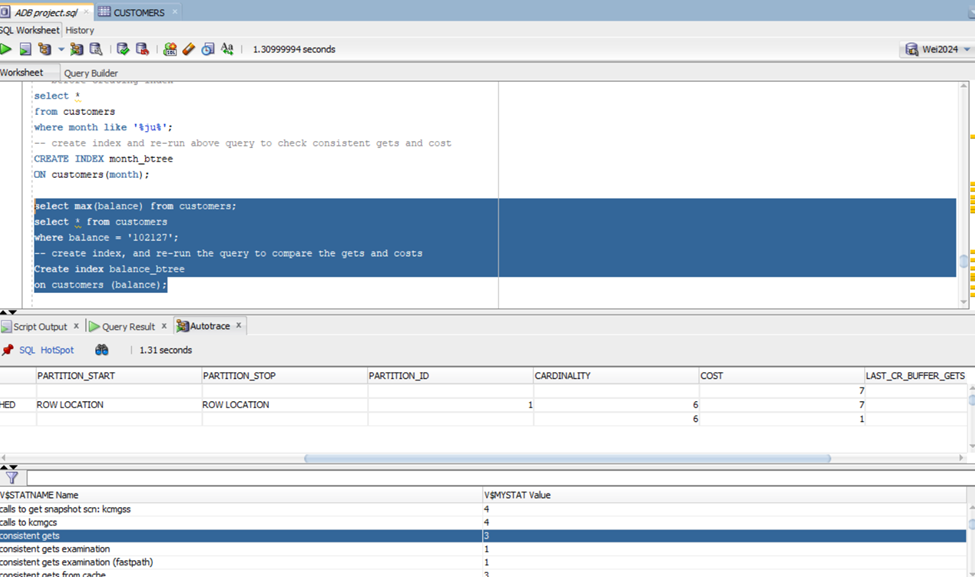
ON customers(month);

CREATE INDEX balance\_btree

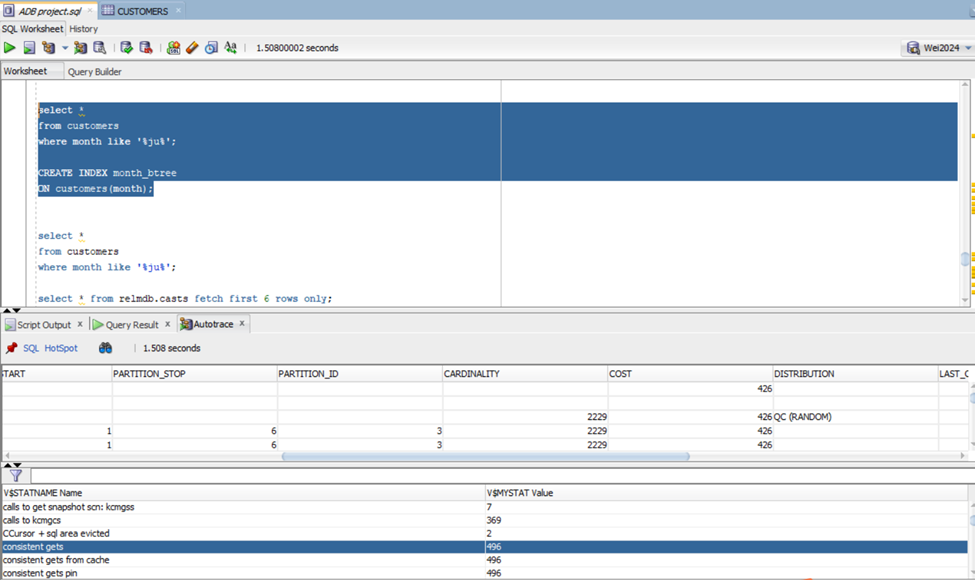
ON customers(balance);

While indexes typically enhance read performance by reducing disk I/O and query execution time, it is important to investigate the improvement in performance. This is because maintaining index structures leads to overhead cost, which might overshadow the improvement if it was not considerable. To check on this, we ran the same queries discussed in Part 2 again using the indexed columns.

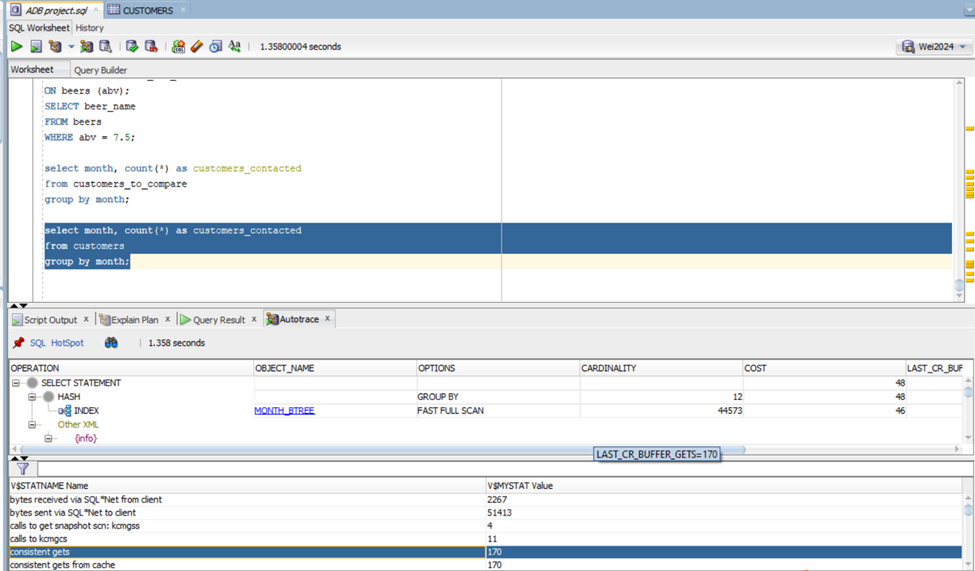
### Point query with indexed column



### Range query with indexed column



### Scan query with indexed column



As we can see, after indexing our database, the performance improves significantly. The point query sees a drastic change in consistent gets as the number goes from 2362 to just 3, as well as the estimate cost. This is because the query uses the indexed column (BALANCE) in its WHERE clause, which helps locate the row much faster than having to execute a full table scan. The range query also returns a lower consistent gets, going down from 540 to 496. The improvement is not as noticeable as seen from the point or scan query, which reports a consistent gets of only 170 compared to 861 without index. As expected, we can see the improvement in performance across all query types, further showing that indexing is effective and necessary for us to achieve an optimal database system.

## 3.3. Parallelism

-- Query to test parallel execution

SELECT customer\_id, balance, results, job\_type

FROM customers

JOIN jobs USING (job\_code)

JOIN job\_type USING (job\_type\_code)

WHERE (age between 30 and 50)

AND balance >= 10000

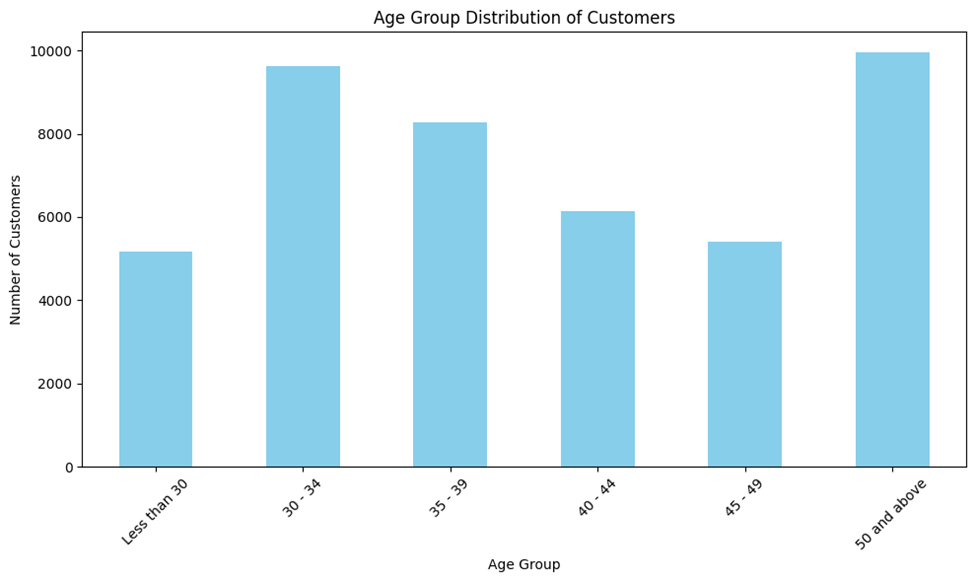
-- Parallel with degree of 4 on and rerun the query

ALTER TABLE customers PARALLEL 4

The last optimization technique we applied in this project is parallelism. Parallelism allows a complex query to be broken down and processed by multiple processors simultaneously, thus improving response time. As shown above: the execution time reduced from 0.11 seconds to 0.107 seconds. In our relatively small database, the execution time change is not obvious. There are likely to be more operations (like IO), but these happen in parallel. However, the virtual server we were using does not really have multiple CPUs - so there is no real parallelism.

# Part 4: Additional Topic

## 4.1. Data Visualization



To generate graphical visualizations like histograms directly within the tool, we imported the data into Python and constructed a figure using the matplotlib package. In the graph above, we used the Customers data and the age column to examine the distribution of the bank’s customers using age groups. Apparently, most of their clients are between 30 and 49 years old, which is understandable because people within this range tend to have more assets for savings and investments. Also, interestingly, between 30 – 49, the number of customers decreases as age increases. Furthermore, people who are 50 and above account for a significant percentage of their customer base.

With SQL developer, we also generated a table visualization with age groups and corresponding group counts as below:

A screenshot of a computer

Description automatically generated

## 4.2. Detecting Outliers – Data Mining

In statistics, an outlier is a data point that differs significantly from other observations. Any observations that fall outside of three standard deviations from the mean are considered an outlier. Having too many outliers in a data set potentially leads to quality issues, creates unusual patterns, and results in misleading insights in analysis. Therefore, we used both python and SQL developer to calculate mean and standard deviation of the customer age, then identify outliers by comparing the data with the 3 standard deviation thresholds. A total number of 330 outliers are detected, using the code and the printout below.

# Python:

# Assuming 'df' is DataFrame containing the 'customers' table

# Calculate mean and standard deviation of age

mean\_age = df['AGE'].mean()

std\_dev\_age = df['AGE'].std()

# Define outlier criteria (e.g., values beyond 3 standard deviations from the mean)

lower\_bound = mean\_age - 3 \* std\_dev\_age

upper\_bound = mean\_age + 3 \* std\_dev\_age

# Identify outliers

outliers = df[(df['AGE'] < lower\_bound) | (df['AGE'] > upper\_bound)]

# Display outliers

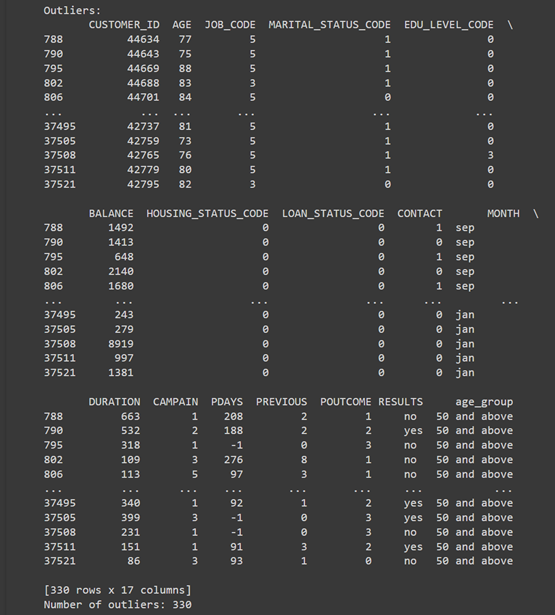
print("Outliers:")

print(outliers)

# Counting the number of outliers

num\_outliers = len(outliers)

print(f"Number of outliers: {num\_outliers}")



SQL developer:

We made use of SQL built-in functions to find the mean (average), standard deviation (stddev), selected the records with balance meeting requirements, and list all the outliers as queries shown below with first thirteen rows:

A screenshot of a computer

Description automatically generated

## 4.3. DBA scripts

There are various scripts that can be used for performance tuning in SQL developer. While executing scripts on a database can have serious consequences and should be done carefully and preferably under the supervision of a DBA (database administrator). Bew are some examples of our scripts commonly used in performance tuning mentioned above:

*CREATE TABLE Customers*

*(*

*Customer\_ID int not null primary key,*

*age NUMBER not null,*

*job\_code int not null,*

*marital\_status\_code int not null,*

*edu\_level\_code int not null,*

*balance int not null,*

*housing\_status\_code int not null,*

*loan\_status\_code int not null,*

*contact int not null,*

*month char(10) not null,*

*duration int not null,*

*campain int not null,*

*pdays int not null,*

*previous int not null,*

*poutcome int not null,*

*results char(3) not null*

*)*

*PARTITION BY RANGE (age)*

*(*

*PARTITION p1 VALUES less than (30),*

*PARTITION p2 VALUES less than (35),*

*PARTITION p3 VALUES less than (40),*

*PARTITION p4 VALUES less than (45),*

*PARTITION p5 VALUES less than (50),*

*PARTITION p6 VALUES less than (95)*

*);*

We created a new table named customers with constraint of primary key customer\_id, data type, data length, not null for data accuracy, and created the partition table here by age range for an improved performance proved above. There are some foreign keys like the edu\_level\_code, we used UI to set the constraints.

*CREATE INDEX month\_btree*

*ON customers(month);*

We also created B-trees like the one shown in the month column to improve database performance by facilitating quick data retrieval and efficient query processing. When we set indexes on specific columns in a database table, we essentially create a data structure that allows the database management system to locate rows quickly using the indexed column values.

*alter table customers parallel 4;*

We enabled parallel execution by setting the PARALLEL degree of the customers table to four. This allowed complex queries to be executed simultaneously across multiple CPUs, marginally reducing response times.