



Database Project: Customer Data for Targeting in the Banking Sector

ISM 6218: Advanced Database Management
Professor Don Berdnt



APRIL 26, 2024

Gang Wei, Zahid Rahman, Asif Imtiaz, Long Nguyen

Table of Contents

Executive Summary	2
Part 1: Database Design.....	4
1.1. Conceptual Design.....	4
1.2. Entity-Relationship Diagram (ERD)	6
Part 2: Query Writing	8
2.1. Point Query	8
2.2. Range Query	9
2.3. Scan Query.....	9
Part 3: Performance Tuning	10
3.1. Partitioning.....	10
3.2. Indexing	12
a. Point query with indexed column.....	12
b. Range query with indexed column	13
c. Scan query with indexed column.....	13
3.3. Parallelism	13
Part 4: Additional Topic	14
4.1. Data Visualization.....	14
4.2. Detecting Outliers – Data Mining	16
4.3. DBA scripts.....	18

Executive Summary

As customers in the banking sector have more access to financial resources and services, they are becoming more aware of marketing tactics from financial institutions. 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 in order to allocate their resources efficiently for the best gains. 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.

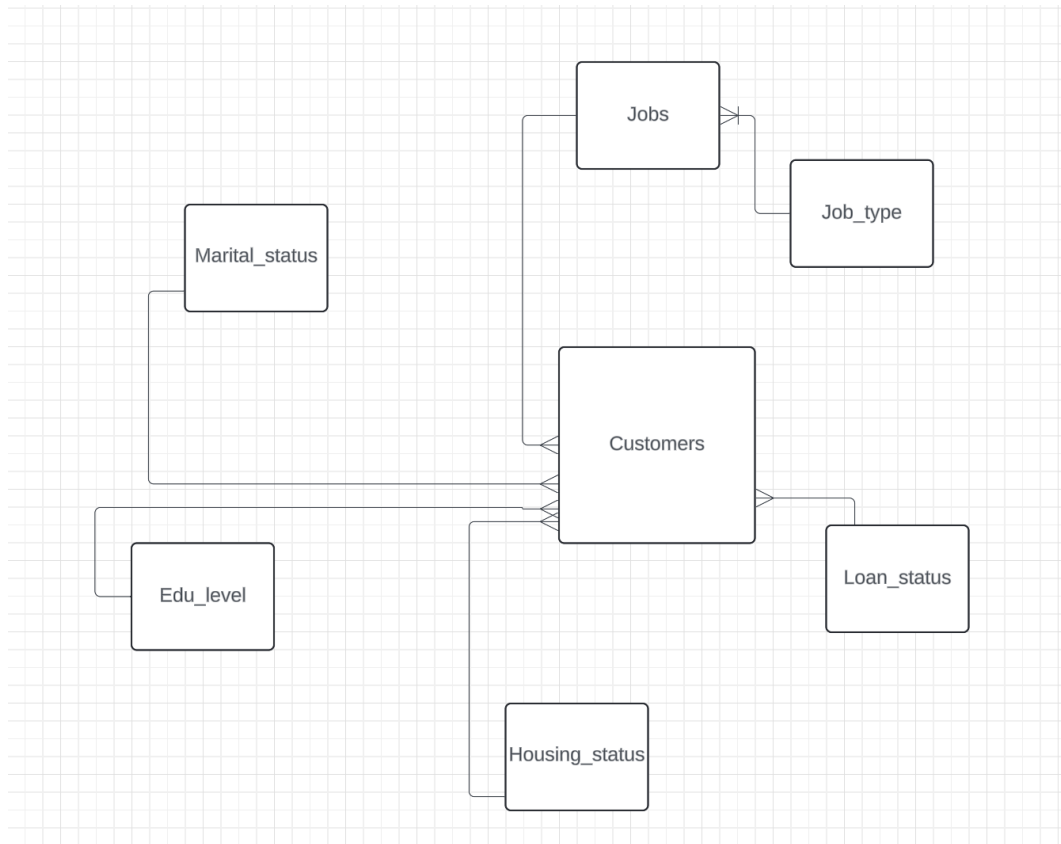
For the sake of this project, we have used the data from the UCI Machine Learning Repository to distribute our database. It would be challenging to collect a large amount of data, which is necessary for some query executions, by us in a brief time. The data is a report of direct telemarketing campaigns 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. We will discuss more about the database design in part 1.

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

1.1. Conceptual Design



Let us start with the conceptual design of the entity-relationship diagrams (ERD), giving 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. Details about each entity are explained in the data dictionary below:

Data Dictionary

Customers

Column	Data Type	Key	Description	Constraints
Customer_ID	int	PK	Unique identifier for each customer	Not null, unique
age	NUMBER		Age of the customer	Not null
job_code	int	FK	Code for the customer's job	Not null, references Jobs(job_code)
marital_status_code	int	FK	Code for the customer's marital status	Not null, references Marital_status(code)
edu_level_code	int	FK	Code for the customer's education level	Not null, references Edu_level(code)
balance	int		Account balance	Not null
housing_status_code	int	FK	Code for the customer's housing status	Not null, references Housing_status(code)
loan_status_code	int	FK	Code indicating if the customer has a loan	Not null, references Loan_status(code)
contact	int		Method of contact with the customer	Not null
month	char(10)		Month of the last contact with the customer	Not null
duration	int		Duration of the last contact in seconds	Not null
campaign	int		Number of contacts performed during campaign	Not null
pdays	int		Days since last campaign contact	Not null
previous	int		Number of contacts before this campaign	Not null
poutcome	int		Outcome of the previous campaign	Not null
results	char(3)		Customer subscription status ('yes'/'no')	Not null

Jobs

Column	Data Type	Key	Description	Constraints
job_code	int	PK	Unique code for each job	Not null, unique
job_name	char(20)		Name of the job	Not null
job_type_code	int	FK	Code for the type of job	Not null, references Job_type(job_type_code)

Job_type

Column	Data Type	Key	Description	Constraints
job_type_code	int	PK	Unique code for each job type	Not null, unique
job_type	char(20)		Description of the job type	Not null

Marital_status

Column	Data Type	Key	Description	Constraints
marital_status_code	int	PK	Unique code for each marital status	Not null, unique
marital_status	char(20)		Description of the marital status	Not null

Edu_level

Column	Data Type	Key	Description	Constraints
edu_level_code	int	PK	Unique code for each education level	Not null, unique
edu_level	char(20)		Description of the education level	Not null

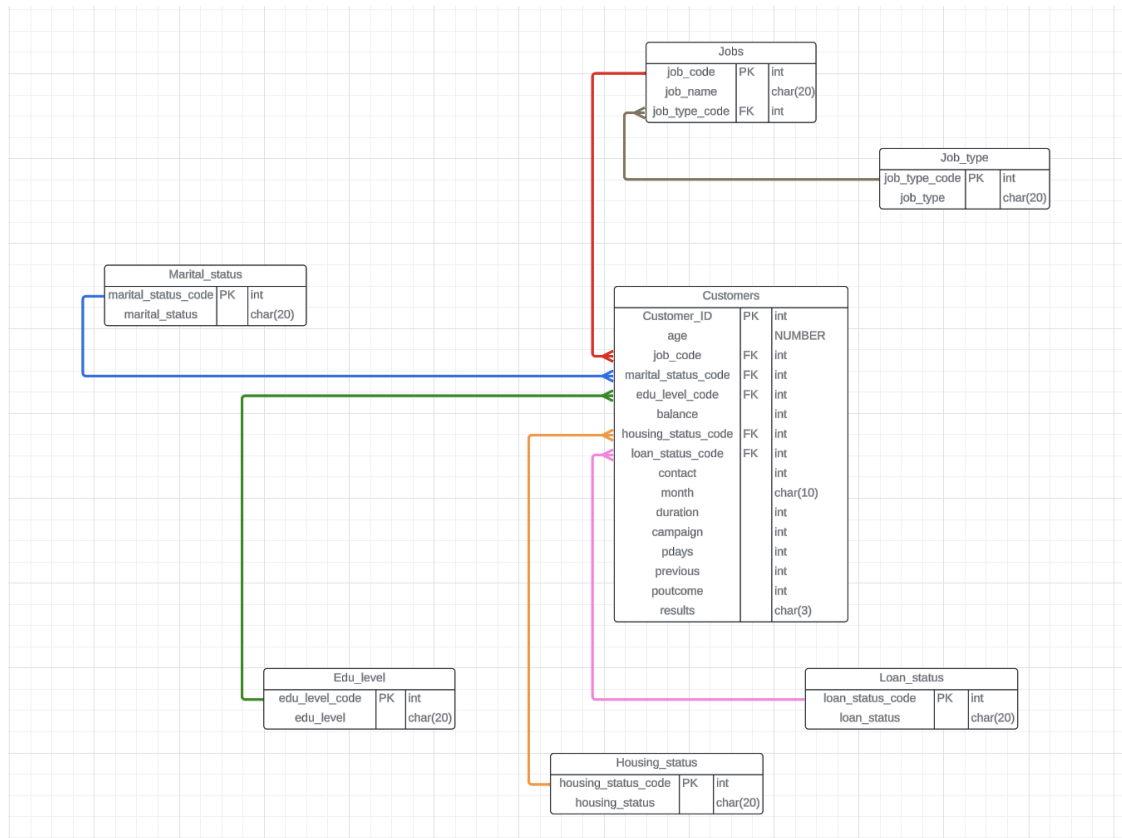
Housing_status

Column	Data Type	Key	Description	Constraints
housing_status_code	int	PK	Unique code for each housing status	Not null, unique
housing_status	char(20)		Description of the housing status	Not null

Loan_status

Column	Data Type	Key	Description	Constraints
loan_status_code	int	PK	Unique code for each loan status	Not null, unique
loan_status	char(20)		Description of the loan status	Not null

1.2.Entity-Relationship Diagram (ERD)



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.

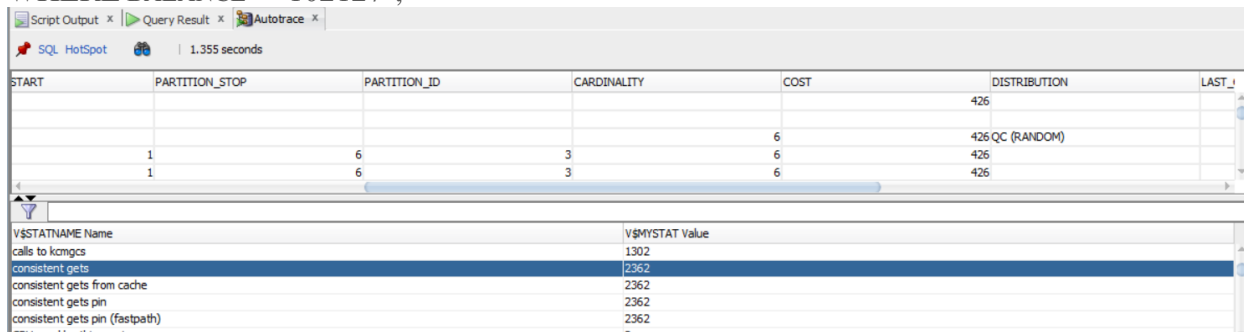
CONSTRAINT_NAME	CONSTRAINT_TYPE	SEARCH_CONDITION	R_OWNER	R_TABLE_NAME	R_CONSTRAINT_NAME
1 EDU_LEVEL_CODE_FK	Foreign_Key	(null)	SQL222	EDU_LEVEL	SYS_C00159137
2 HOUSING_STATUS_CODE_FK	Foreign_Key	(null)	SQL222	HOUSING_STATUS	SYS_C00158944
3 JOB_CODE_FK	Foreign_Key	(null)	SQL222	JOBS	SYS_C00158955
4 LOAN_STATUS_CODE_FK	Foreign_Key	(null)	SQL222	LOAN_STATUS	SYS_C00158947
5 MARITAL_STATUS_CODE_FK	Foreign_Key	(null)	SQL222	MARITAL_STATUS	SYS_C00158935

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';
```



The screenshot shows the SQL Developer interface. The top pane displays the query results for the SQL statement. The bottom pane shows the execution statistics.

START	PARTITION_STOP	PARTITION_ID	CARDINALITY	COST	DISTRIBUTION	LAST...
					426	
				6	426 QC (RANDOM)	
1		6	3	6	426	
1		6	3	6	426	

V\$STATNAME Name	V\$MYSTAT Value
calls to kcmgcs	1302
consistent gets	2362
consistent gets from cache	2362
consistent gets pin	2362
consistent gets pin (fastpath)	2362

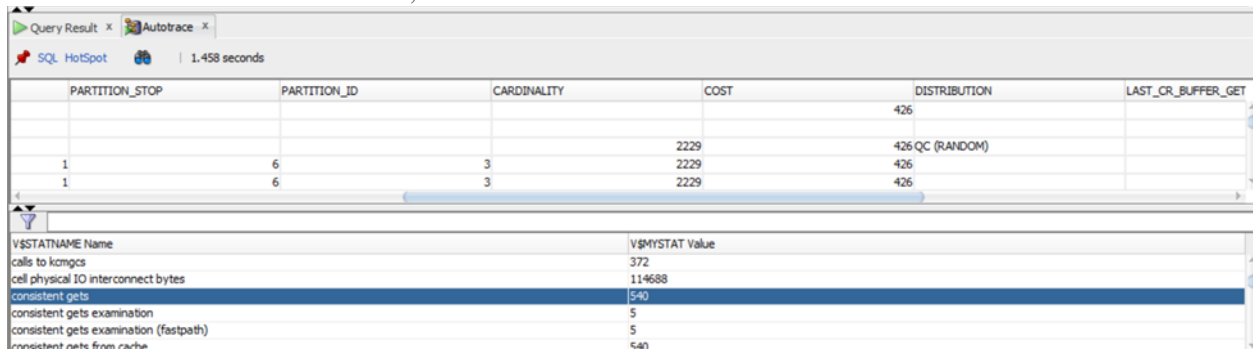
In part 2, we will begin writing some queries to see if the database functions appropriately, starting with a point query identifying the customer with the highest balance in their account. This query consists of two parts. The first part calculates the maximum BALANCE value from the CUSTOMERS table, 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 screenshot shows the Autotrace window for a query. The top table displays execution statistics for three partitions. The bottom table shows various V\$STATNAME statistics.

PARTITION_STOP	PARTITION_ID	CARDINALITY	COST	DISTRIBUTION	LAST_CR_BUFFER_GET
				426	
1	6	3	2229	426 QC (RANDOM)	
1	6	3	2229	426	

V\$STATNAME Name	V\$STATNAME Value
calls to komgs	372
cell physical IO interconnect bytes	114688
consistent gets	540
consistent gets examination	5
consistent gets examination (fastpath)	5
inconsistent reads from cache	640

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 screenshot shows the SQL Developer Autotrace window. The top section displays the execution plan for a query, with the following details:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	DISTRIBUTION
SELECT STATEMENT					39
PX COORDINATOR					
PX SEND	SYS.TQ10001	QC (RANDOM)		12	39 QC (RANDOM)
HASH		GROUP BY		12	39
PX RECEIVE				12	39

The bottom section shows the V\$STATNAME statistics for the query execution:

V\$STATNAME Name	V\$STATVALUE Value
bytes received via SQL*Net from client	2277
bytes sent via SQL*Net to client	52586
calls to get snapshot scn: kcmgss	4
calls to kcmgcs	433
consistent gets	861
consistent gets from cache	861

The last query we ran was a range 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)
);

	PARTITION_NAME	LAST_ANALYZED	NUM_ROWS	BLOCKS	SAMPLE_SIZE	HIGH_VALUE
1	P6	22-APR-24	9945	1006	9945	95
2	P5	22-APR-24	5417	1006	5417	50
3	P4	22-APR-24	6139	1006	6139	45
4	P3	22-APR-24	8265	1006	8265	40
5	P2	22-APR-24	9631	1006	9631	35
6	P1	22-APR-24	5176	628	5176	30

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;
```

OPERATION	OBJECT_NAME	OPTIONS	PARTITION_START	PARTITION_STOP	PARTITION_ID
SELECT STATEMENT					
PX COORDINATOR					
PX SEND	SYS.TQ10000	QC (RANDOM)			
PX BLOCK		ITERATOR		3	3
TABLE ACCESS	CUSTOMERS	FULL		3	3

V\$STATNAME Name	V\$MYSTAT Value
cell physical IO interconnect bytes	114688
consistent gets	1655
consistent gets examination	3
consistent gets examination (fastpath)	3
consistent gets from cache	1655
consistent gets pin	1652

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	DISTRIBUTION
SELECT STATEMENT					38
PX COORDINATOR					
PX SEND	SYS.TQ10000	QC (RANDOM)		1873	38 QC (RANDOM)
PX BLOCK		ITERATOR		1873	38
TABLE ACCESS	CUSTOMERS_TO_COMPARE	FULL		1873	38

V\$STATNAME Name	V\$MYSTAT Value
bytes received via SQL*Net from client	2266
bytes sent via SQL*Net to client	52193
calls to get snapshot scn: kcmgcs	4
calls to kcmgcs	425
consistent gets	361
consistent gets from cache	861

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.

a. Point query with indexed column

PARTITION_START	PARTITION_STOP	PARTITION_ID	CARDINALITY	COST	LAST_CR_BUFFER_GETS
HED	ROW LOCATION	ROW LOCATION	1	6	7
				6	7
					1

V\$STATNAME Name	V\$MYSTAT Value
calls to get snapshot scn: kcmgcs	4
calls to kcmgcs	4
consistent gets	3
consistent gets examination	1
consistent gets examination (fastpath)	1
consistent reads from cache	3

b. Range query with indexed column

START	PARTITION_STOP	PARTITION_ID	CARDINALITY	COST	DISTRIBUTION	LAST_C
					426	
1	6	3	2229	426 QC (RANDOM)		
1	6	3	2229	426		

V\$STATNAME Name	V\$MYSTAT Value
calls to get snapshot scn: kcmgss	7
calls to kcmgcs	369
CCursor + sql area evicted	2
consistent gets	496
consistent gets from cache	496
consistent gets pin	496

c. Scan query with indexed column

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUF
SELECT STATEMENT					48
HASH		GROUP BY		12	48
INDEX	MONTH_ITREE	FAST FULL SCAN		44573	46

V\$STATNAME Name	V\$MYSTAT Value
bytes received via SQL*Net from client	2267
bytes sent via SQL*Net to client	51413
calls to get snapshot scn: kcmgss	4
calls to kcmgcs	11
consistent gets	170
consistent gets from cache	170

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

```

```

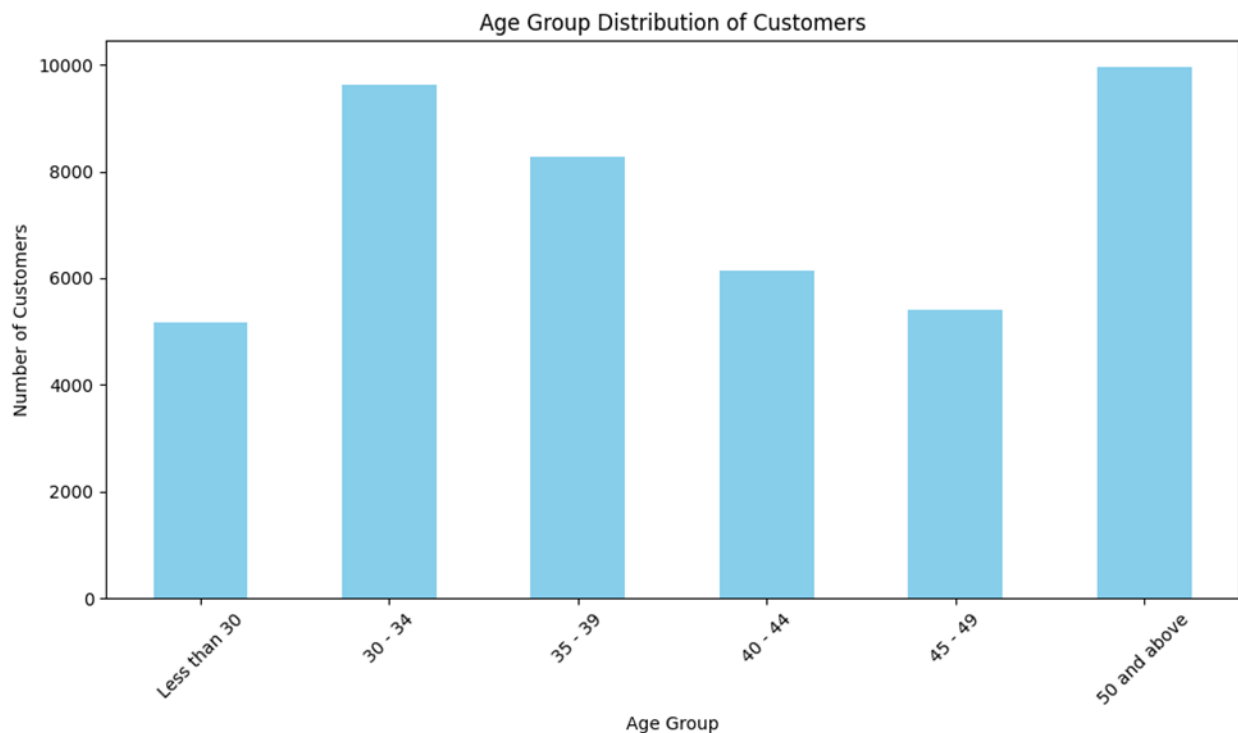
-- PARALLEL WITH DEGREE OF 4 ON AND RERUN THE QUERY
ALTER TABLE CUSTOMERS PARALLEL 4

```

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.

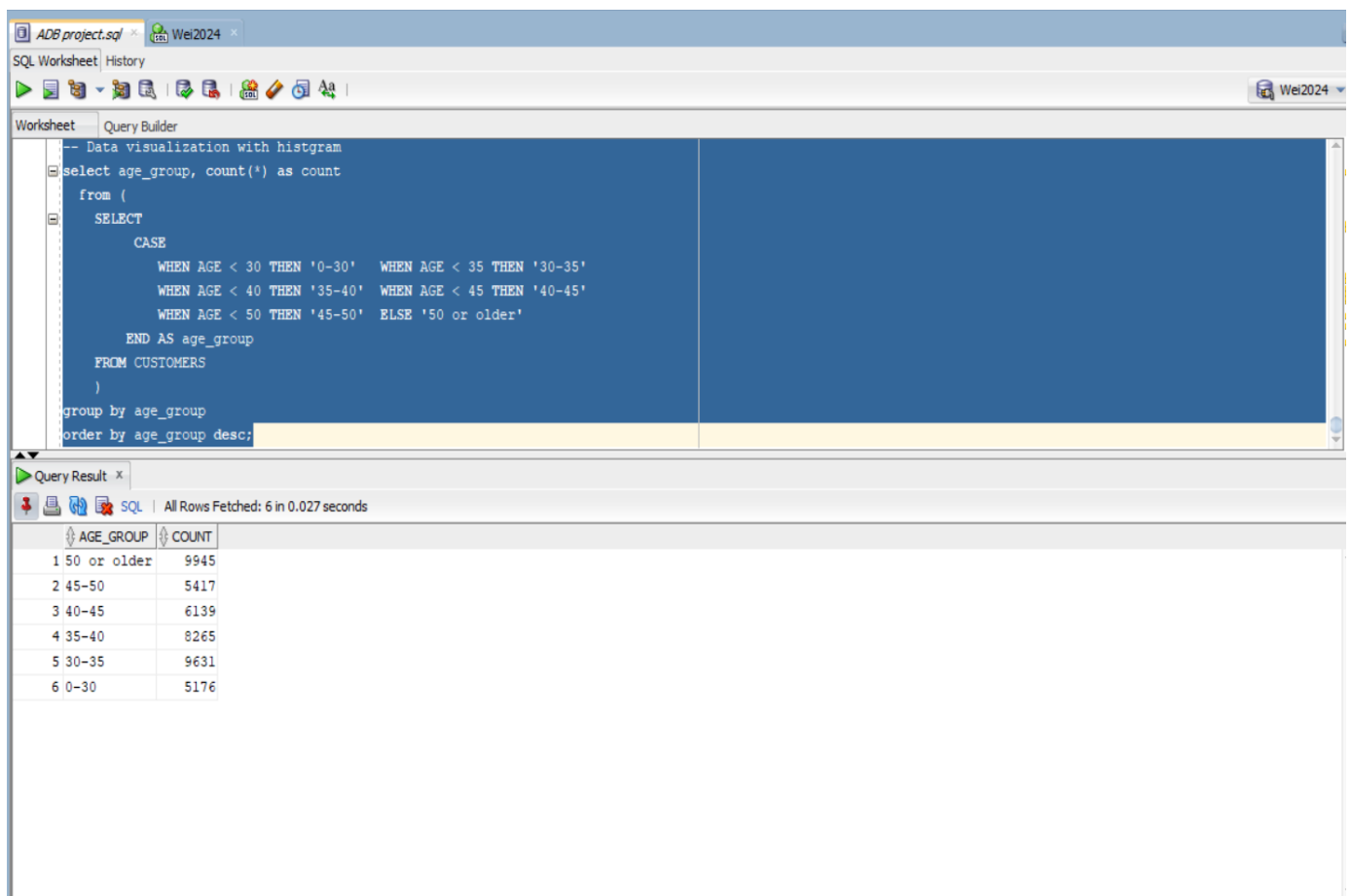
Part 4: Additional Topic

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:



The screenshot shows the SQL Developer interface with a query window and a results window. The query window contains the following SQL code:

```
-- Data visualization with histogram
select age_group, count(*) as count
from (
  SELECT
    CASE
      WHEN AGE < 30 THEN '0-30' WHEN AGE < 35 THEN '30-35'
      WHEN AGE < 40 THEN '35-40' WHEN AGE < 45 THEN '40-45'
      WHEN AGE < 50 THEN '45-50' ELSE '50 or older'
    END AS age_group
  FROM CUSTOMERS
)
group by age_group
order by age_group desc;
```

The results window shows the following table:

AGE_GROUP	COUNT
1 50 or older	9945
2 45-50	5417
3 40-45	6139
4 35-40	8265
5 30-35	9631
6 0-30	5176

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}')
```

Outliers:						
	CUSTOMER_ID	AGE	JOB_CODE	MARITAL_STATUS_CODE	EDU_LEVEL_CODE	\
788	44634	77	5	1	0	
790	44643	75	5	1	0	
795	44669	88	5	1	0	
802	44688	83	3	1	0	
806	44701	84	5	0	0	
...
37495	42737	81	5	1	0	
37505	42759	73	5	1	0	
37508	42765	76	5	1	3	
37511	42779	80	5	1	0	
37521	42795	82	3	0	0	

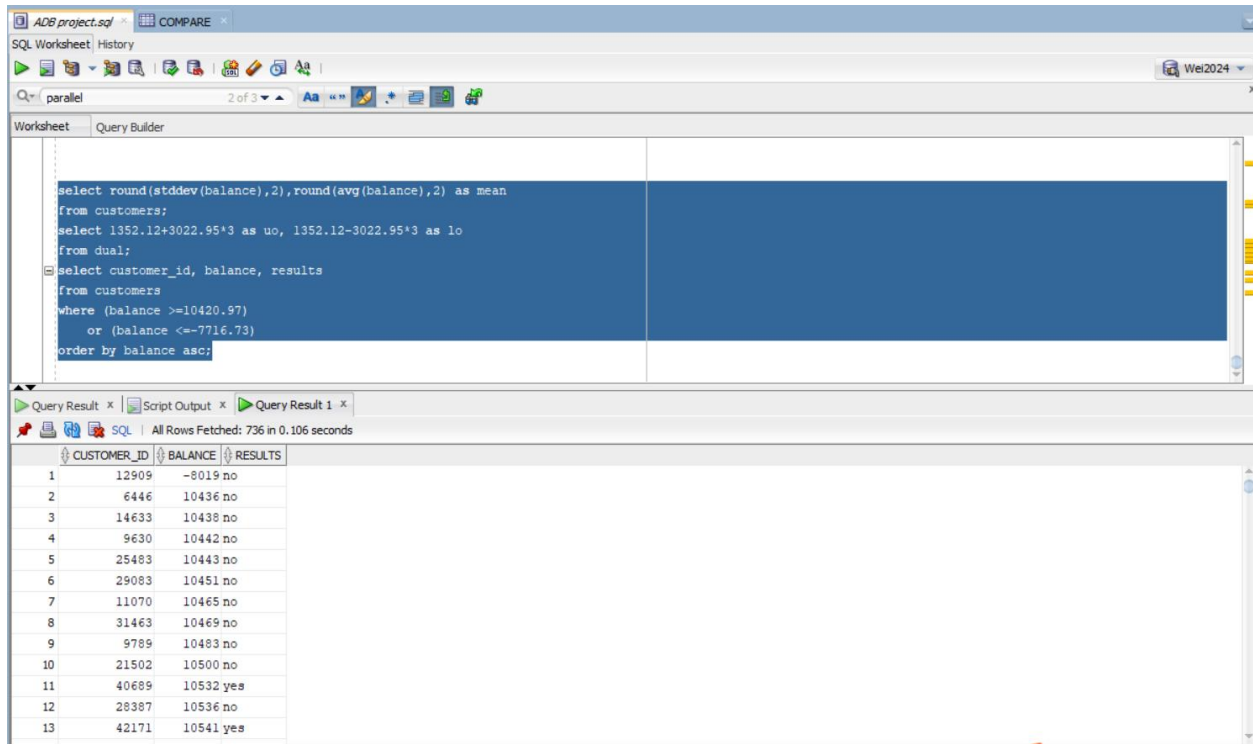
	BALANCE	HOUSING_STATUS_CODE	LOAN_STATUS_CODE	CONTACT	MONTH	\
788	1492	0	0	1	sep	
790	1413	0	0	0	sep	
795	648	0	0	1	sep	
802	2140	0	0	0	sep	
806	1680	0	0	1	sep	
...
37495	243	0	0	0	jan	
37505	279	0	0	0	jan	
37508	8919	0	0	0	jan	
37511	997	0	0	0	jan	
37521	1381	0	0	0	jan	

	DURATION	CAMPAIN	PDAYS	PREVIOUS	POUTCOME	RESULTS	age_group
788	663	1	208	2	1	no	50 and above
790	532	2	188	2	2	yes	50 and above
795	318	1	-1	0	3	no	50 and above
802	109	3	276	8	1	no	50 and above
806	113	5	97	3	1	no	50 and above
...
37495	340	1	92	1	2	yes	50 and above
37505	399	3	-1	0	3	yes	50 and above
37508	231	1	-1	0	3	no	50 and above
37511	151	1	91	3	2	yes	50 and above
37521	86	3	93	1	0	no	50 and above

[330 rows x 17 columns]
Number of outliers: 330

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:



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). Below 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,
  campaign 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.