Online Retailing and Shopping: An Academical Simulation to Databases

António Lima, Jozef Stasko, Yu-Lin Wang
UNIZA – University of Zilina
Zilina, Slovakia
resende_lima@stud.uniza.sk, stasko3@stud.uniza.sk, wang2@stud.uniza.sk

Teodora Gavrilović
University of Belgrade
Belgrade, Serbia
tg20213005@student.fon.bg.ac.rs

***Abstract*— How is a Database for Online Shopping? What are good practices in generating and managing the database? The following essay has the purpose to serve as an academical paper, where students from the University of Belgrade, Serbia and from the UNIZA – University of Zilina worked together to create and develop a model for Online Shopping websites and E-Commerce, giving examples of all the process.**

The process started with the creation of our Relational Model. It was necessary to create and define our tables, needed according our purpose, a database for Online retailing. With this principle, they were defined 9 tables which will contain different attributes and registrations. The tables created are: *Customer, Image, Invoice, OrderMain, OrderProduct, OrderStatus, PaymentType, Product and PromoCode*. With this information, it's possible to simulate a full buying process and generate different registrations to the database.

The following step was to generate our data. This was possible with procedures created to all of the tables with automatic data generation. The only table that didn't have these random values was the table Image, which the information as web scrapped from a supermarket Tesco Database. With the way we're generating data, a procedure was also developed where we could update instantly our table OrderMain with the price of each order. There was the consideration of transforming URL data from VARCHAR to BLOB. For the table Customer, the passwords were encrypted via the DBMS_CRYPTO package.

The model development and implementation included also the task of developing Partitioning for this model was considered because of the size of the database. Our table OrderProduct contains a lot of big amount of data and so it was necessary this type of division, making our database more threated and distributed. For our index's creation and processes, the were defined 5 indexes called: ExpensiveOrder, two NumberOfOrder (depending on the following tables: OrderMain and OrderStatus, PromoCode and PromoCodeExpiration.

Our number of indexes depend on the queries that we developed to the simulation process of our database. These queries will be part of our process for inspecting and understanding how the behavior of the model is developed. They consist in: Number of orders in first quartals of years where order status is payed; The customer with the most expensive order (the biggest amount) grouped by payment type; Top 10 most expensive products; Top 10 orders with the smallest amount to be payed; The order with the most products (overall products not product types); Customer with the most promo codes that give more than 50% discount; Promo codes grouped by % (<20, >20&<40, >40); All promo codes which will be expired by February 2022; Percentage of orders where order status is returned and Top 10 product with most orders

Based on the model review done we also made some changes to the queries, so that we could understand new database distributions. This process was important was critical to understand that we were only working with databases where there was a maximum of 10 products in an order.

With these queries we could easily understand and compare how a system of online shopping should work. This part is when we enter the model review, where we try to test our database and understand the results. The type of data that was generated also tried to diversify and be more accurate. In was in this part where we tested and understood the database, compared the difference (after our partitioning was done). For example, we could understand that our question number 10 could not improve that much was subjected to the partitioning process.

The costs of the different questions helped us identify how the information that we have could affect the normality of our database. It is possible so to understand which of the queries also has the lowest cost and requires less processing. For example, that our questions 2 and 6 assume their role as the cheapest. These process executions are easier when compared with our other questions, also because of the amount of data he's iterating over to give the result.

There is a limitation where the database doesn't reflect the spent PromoCodes. The only performance that was totally measured is related to the queries that were made, their cardinality and costs. Although, it was also considered the chance of more statistical approaches to the database.

One of our last processes was reviewing what was done, what we learned and what possible aspects to improve on developing this type of database, presentng all this reflection on this academic paper.

I. INTRODUCTION

The following article is created on the context of the course of Advanced Database Management, from the Faculty of Management and Informatics, UNIZA - University of Zilina and Database II course on Information System Module from the Faculty of Organizational Sciences, University of Belgrade.

We were proposed by our teachers to create and develop a database about any theme we wanted. The group decided to pick the online shopping topic, as this area seemed quite challenging and increased their businesses with the current pandemic. According to analysis order data, we can understand more about our customers and find potential business.

As a way of developing our knowledge about the Database topic, we used some of the theory that we learned during our course and practiced by the realization of this project. We were challenged to use some advanced database techniques that contributed to the development of this article.

After creating and designing the relational model with different tables and attributes, we developed this model in Oracle SQL Server where we created the classes and then uploaded the data. This last process was done by using Procedures, a subroutine that allows the data insertion in a simplified way.

II. MODEL DEVELOPMENT

A. Model design

Our first operational process was the creation and design of our database model. In this part we started by deciding the number of tables, what attributes they would contain and the relationship between them (by also reviewing similar databases that would exist online).

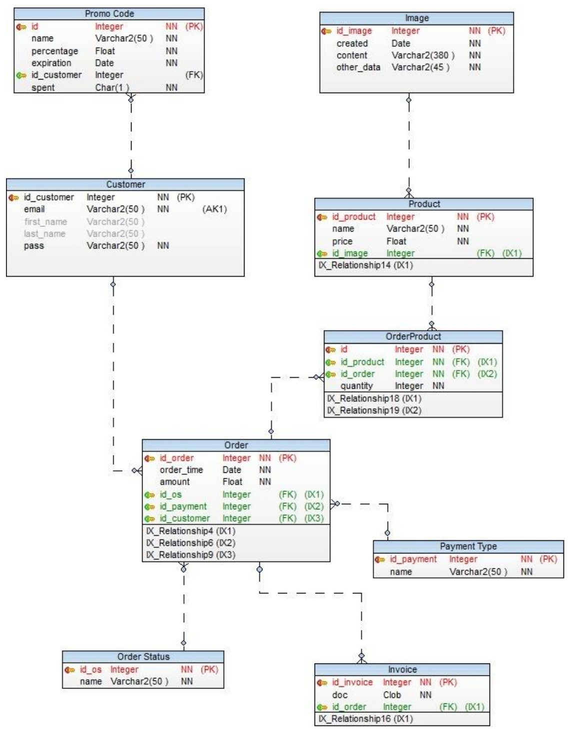


Figure 1 - Relational Model

Our number of tables consist of 9, being *Customer*, *Image*, *Invoice*, *OrderMain*, *OrderProduct*, *OrderStatus*, *PaymentType*, *Product and PromoCode*.

1. The table ***Customer*** is composed by the attributes: *id_customer* (PK[1]), *email*, *first_name*, last*_name* and *pass*.
2. The table ***Image*** is composed by the attributes: *id_image*, *created*, *content* and *other_data*.
3. The table ***Invoice*** contains the attributes: id*_invoice*, *doc* and *id_order* (FK[2] to the table *Order*).
4. The table ***OrderMain*** is composed by the attributes: *id_order* (PK), *order_time*, *amount*, *id_os* (FK to the table *OrderStatus*, *id_payment* (FK to the table *PaymentType* and *id_customer* (FK to the table *Customer*).
5. The table ***OrderProduct*** has the following attributes: *id* (PK), *id_product* (FK to the table *Product*), *id_order* (FK to table *OrderMain*) and *quantity*.
6. The table ***OrderStatus*** is composed by the attributes: *id_os* (PK) and *name*.
7. The table ***PaymentType*** has the attributes: *id_payment* (PK) and *name*.
8. The table ***Product*** contains the attributes: *id_product* (PK), *name*, *price* and *id_image* (FK to the table *Image*).
9. The table ***PromoCode*** has the attributes: *id* (PK), *name*, *percentage*, *expiration*, *id_customer* (FK to the table *Customer*) and *spent*.

[1] PK – Primary Key

[2] FK – Foreign Key

B. Model insertion and implementation

(1)Insert data

Related with the model insertion and implementation, after creating the tables in Oracle SQL Server, the next step was data insertion. For this step, we created multiple Procedures, a subroutine that allows the data insertion in a simplified way. Before we run the procedures, we had an insertion query to generate 100 products to the table *Product* with random prices. For the *Image* table, we inserted the information via a .csv file that we used 100 urls.

The first procedure is called *insert_into_customer* and is related with the table *Customer*, and we generated 10000 users, with sequential id, random email, first name and last name and then, the password. The password was encrypted by using a library from Oracle named *DBMS_CRYPTO*, and the function *hash*. The package *DBMS_CRYPTO* is the correct package to generate hashes. It is not granted to public by default, you will have to grant it specifically (GRANT EXECUTE ON SYS.DBMS_CRYPTO TO user1). The result of this function is of datatype RAW. You can store it in a RAW column or convert it to VARCHAR2 using the RAWTOHEX or UTL_ENCODE.BASE64_ENCODE functions.

The HASH function is overloaded to accept three datatypes as input: RAW, CLOB and BLOB. Due to the rules of implicit conversion, if you use a VARCHAR2 as input, Oracle will try to convert it to RAW and will most likely fail since this conversion only works with hexadecimal strings. If you use VARCHAR2 then, you need to convert the input to a binary datatype or a CLOB.

The following procedure is named *insert_into_promoCode* for the table *PromoCode* and generated 100000 promocodes with sequential id, random customer, name, expiration date and percentage. This expiration date will explain if the promocode is still available for being used or not. The next procedure is named *insert_invoice* for the table *Invoice* and this will insert crossed data from *Customer* and *PaymentType* in the *doc* attribute.

The next procedure is called *insert_into_order* to the table *OrderMain* and will generate 10000 orders with sequential id and different random attributes like, the order status, payment type, the customer and the date. This procedure also inserts information in the *Invoice* table. The following procedure is *insert_into_OrderProduct* and will insert 100000 random values from the table *OrderMain* to the table *OrderProduct*, by generating random values to the *order_id*, *product_id* and the own id.

The last procedure is to *update_OrderMain* and has the purpose of updating the amount for the table *OrderMain*, based on the information from the *OrderProduct* table (information from *Product* table).

(2) Partitioning

The Partitioning Concept, as its name suggests, is based on dividing data into smaller portions (partitions) with the aim of optimization of database. It allows tables and indexes to be divided, enabling these database objects to be managed and accessed at a finer level of granularity. Oracle provides a rich variety of partitioning strategies and extensions to address different requirements.

Given that the database on which this paper is based is characterized by tables of relatively low complexity and large number of records, Horizontal Partitioning stands out as a good approach. This scenario involves creating sub tables within a defined range. The goal is to increase the efficiency of data access.

The table of greatest interest is the OrderProduct which contains 100,000 records. This amount of data makes it an excellent candidate for partitioning. As the table has only the quantity attribute, in addition to the keys, partitions are created above this field. It is necessary to determine the type of criteria for hierarchical partitioning. The Quantity attribute is an integer value, in the range 1 to 10. Therefore, we conclude that it is appropriate to apply the Range criterion. The number of partitions was chosen to be three:

- Small (27957 records)
- Medium (44525 records)
- Big (27518 records)

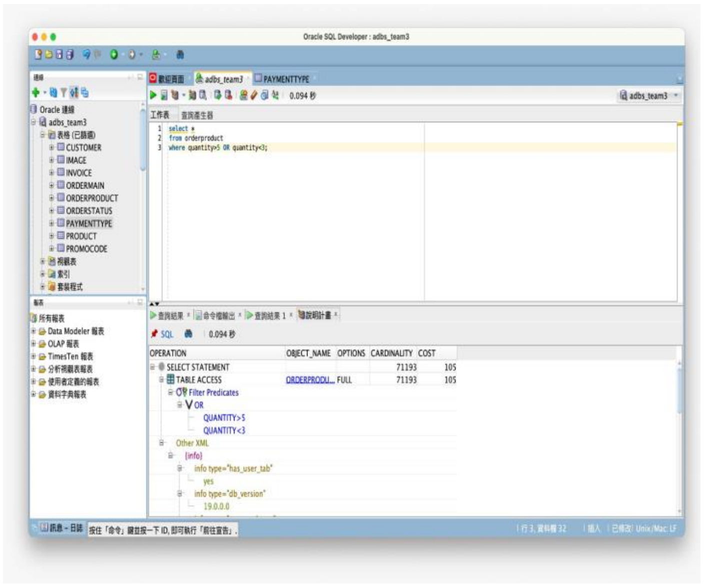
The logical aspect of the model and data structures were analyzed, in order to determine the boundary for each partition. Since it is a question of the quantity of products purchased, it was considered that more than 3 products go beyond the scope of a small purchase. Similarly, over 7 copies of the same product are a large number. Statistics of the generated data also support this choice. It can be seen that the first and third partitions have approximately the same number of records, while the second contains about 60% more. By choosing the boundaries differently, the partitions would not be logical wholes and would be less uniform.

It is important to remember that in this case we are talking about generated data, which, although created as a simulation of a real system, cannot truly mimic it. Therefore, when using real data, it would be necessary to analyze the quantity distribution and change the range if necessary.

Figure 2 shows the execution plan before creating the partitions. Partitioning efficiency is monitored based on queries:

```
select *
from orderproduct
where quantity>5 OR quantity<3;
```

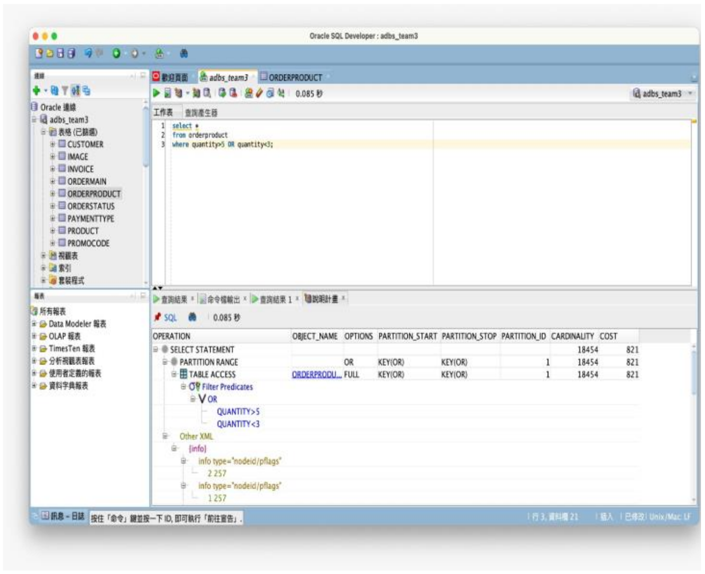
It can be seen that the cost is 105 units, the cardinality is very high, and the execution time is 0.094 sec. Figure 3 shows plan after partitions. Unfortunately, it can be seen that the cost is much higher with partitions. On the other hand, the cardinalities that indicate the number of joined rows are much smaller, and the execution time itself is shorter. This indicates that this partitioning is not suitable for this type of query. A space in which partitions would prove useful would be complex queries that are limited to values from only one partition, i.e., directly access it.



```
select *
from orderproduct
where quantity>5 OR quantity<3;
```

```
OPERATION                OBJECT_NAME     OPTIONS  CARDINALITY  COST
SELECT STATEMENT                                        71193   105
  TABLE ACCESS           ORDERPRODU...   FULL           71193   105
    Filter Predicates
      OR
        QUANTITY>5
        QUANTITY<3
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
```

Figure 2 -- Question 11 without Partitioning



```
select *
from orderproduct
where quantity>5 OR quantity<3;
```

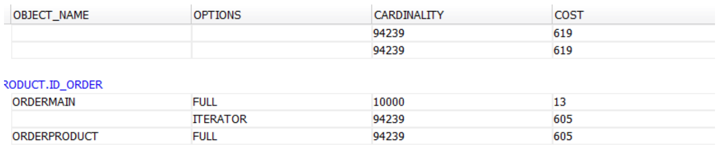
```
OPERATION              OBJECT_NAME    OPTIONS  PARTITION_START  PARTITION_STOP  PARTITION_ID  CARDINALITY  COST
SELECT STATEMENT                                                                                    18454   821
  PARTITION RANGE                     OR       KEY(OR)          KEY(OR)                    1        18454   821
    TABLE ACCESS       ORDERPRODU...  FULL     KEY(OR)          KEY(OR)                    1        18454   821
      Filter Predicates
        OR
          QUANTITY>5
          QUANTITY<3
  Other XML
    {info}
      info type="nodeid/pflags"
        2 257
      info type="nodeid/pflags"
        1 257
```

Figure 3- Question 11 with Partitioning

As can be seen below, accessing data through a partition is much more efficient than directly applying the conditions in the query. This was tested on queries:

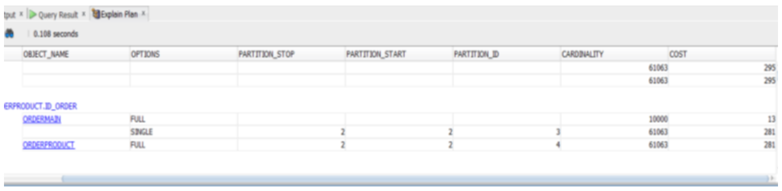
```
select  *  from  OrderMain  join  OrderProduct
partition(medium) using(id_order);

select  *from  OrderMain  join  OrderProduct
using(id_order)
where    OrderProduct.quantity    >3    and
OrderProduct.quantity <8 ;
```



OBJECT_NAME	OPTIONS	CARDINALITY	COST
		94239	619
		94239	619
RODUCT.ID_ORDER			
ORDERMAIN	FULL	10000	13
	ITERATOR	94239	605
ORDERPRODUCT	FULL	94239	605

Figure 4- Question 12 without Pertitioning



0.108 seconds

OBJECT_NAME	OPTIONS	PARTITION_STOP	PARTITION_START	PARTITION_ID	CARDINALITY	COST
					61063	295
					61063	295
ERPRODUCT.ID_ORDER						
ORDERMAIN	FULL				10000	13
	SINGLE	2	2	3	61063	281
ORDERPRODUCT	FULL	2	2	4	61063	281

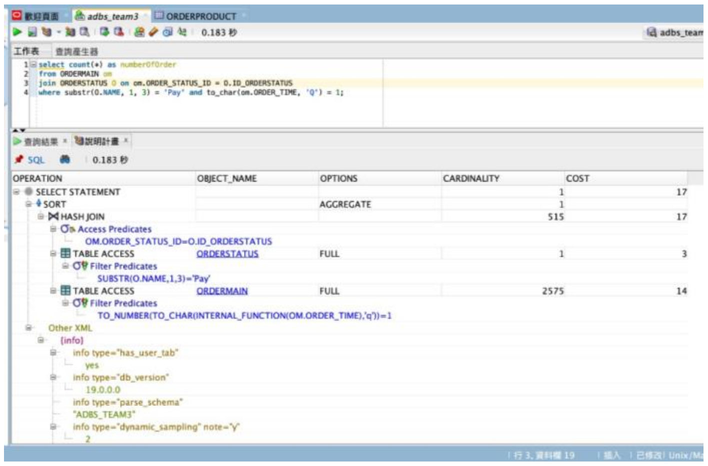
Figure 5- Question 12 with Pertitioning

The conclusion we came to through the partition testing process is that the existence of a large number of records is not a necessary measure of the convenience of partitioning. It is necessary to observe the process from the business side as well. Next, analyze what queries are actually used and the distribution of real data.

(3)Index

How to set Indexes in database depends on queries which will be detailed described in next section (C. Model Review). Here, we focus on how to decide the index and how is the result.

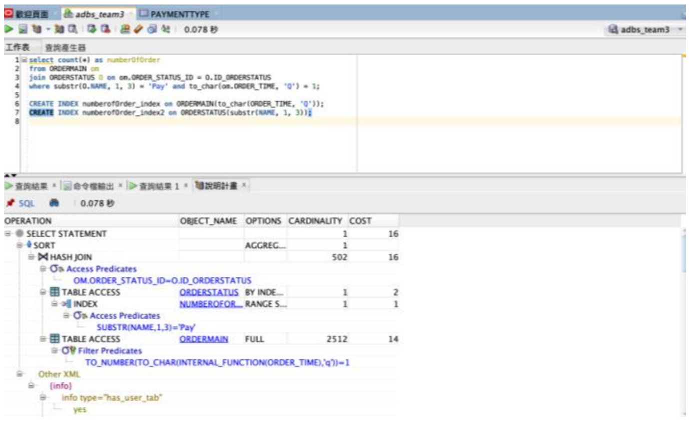
About Question 1, we can find the query spent works on finding first quarter from attribute *ordertime* in Table *OrderMain* and find the payment type from attribute *name* in Table *PaymentType*. Thus, create index in these two attributes to shorten query time. Compare to Figure 6 and Figure 7, can see that running time is 0.105 second less and the works query cost from 17 reduce to 16.



```
select count(*) as numberOfOrder
from ORDERMAIN om
join ORDERSTATUS O on om.ORDER_STATUS_ID = O.ID_ORDERSTATUS
where substr(O.NAME, 1, 3) = 'Pay' and to_char(om.ORDER_TIME, 'Q') = 1;
```

```
OPERATION                  OBJECT_NAME    OPTIONS     CARDINALITY  COST
SELECT STATEMENT                                                1    17
  SORT                                    AGGREGATE             1
    HASH JOIN                                                 515    17
      Access Predicates
        OM.ORDER_STATUS_ID=O.ID_ORDERSTATUS
      TABLE ACCESS         ORDERSTATUS    FULL                  1     3
        Filter Predicates
          SUBSTR(O.NAME,1,3)='Pay'
      TABLE ACCESS         ORDERMAIN      FULL               2575    14
        Filter Predicates
          TO_NUMBER(TO_CHAR(INTERNAL_FUNCTION(OM.ORDER_TIME),'q'))=1
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
      info type="parse_schema"
      "ADBS_TEAM3"
      info type="dynamic_sampling" note="y"
        2
```

Figure 6 - Question 1



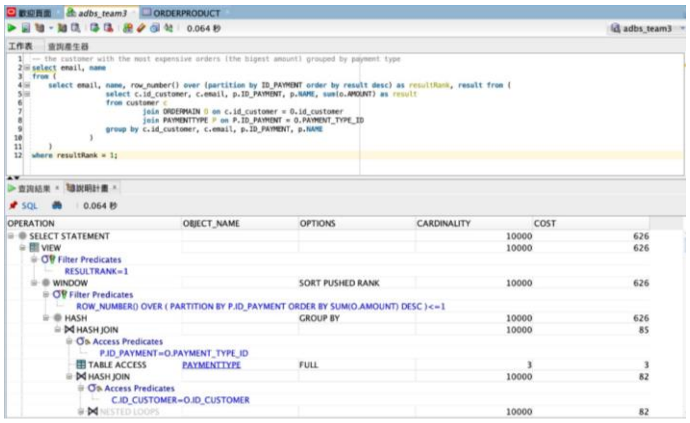
```sql
select count(*) as numberOfOrder
from ORDERMAIN om
join ORDERSTATUS O on om.ORDER_STATUS_ID = O.ID_ORDERSTATUS
where substr(O.NAME, 1, 3) = 'Pay' and to_char(om.ORDER_TIME, 'Q') = 1;

CREATE INDEX numberofOrder_index on ORDERMAIN(to_char(ORDER_TIME, 'Q'));
CREATE INDEX numberofOrder_index2 on ORDERSTATUS(substr(NAME, 1, 3));
```

```
0.078 秒
OPERATION                      OBJECT_NAME    OPTIONS      CARDINALITY  COST
SELECT STATEMENT                                                     1    16
  SORT                                        AGGREG...              1
    HASH JOIN                                                      502    16
      Access Predicates
        OM.ORDER_STATUS_ID=O.ID_ORDERSTATUS
      TABLE ACCESS             ORDERSTATUS    BY INDE...             1     2
        INDEX                  NUMBEROFOR...  RANGE S...             1     1
          Access Predicates
            SUBSTR(NAME,1,3)='Pay'
      TABLE ACCESS             ORDERMAIN      FULL                2512    14
        Filter Predicates
          TO_NUMBER(TO_CHAR(INTERNAL_FUNCTION(ORDER_TIME),'q'))=1
  Other XML
    {info}
      info type="has_user_tab"
        yes
```

Figure 7 - Question 1 Index

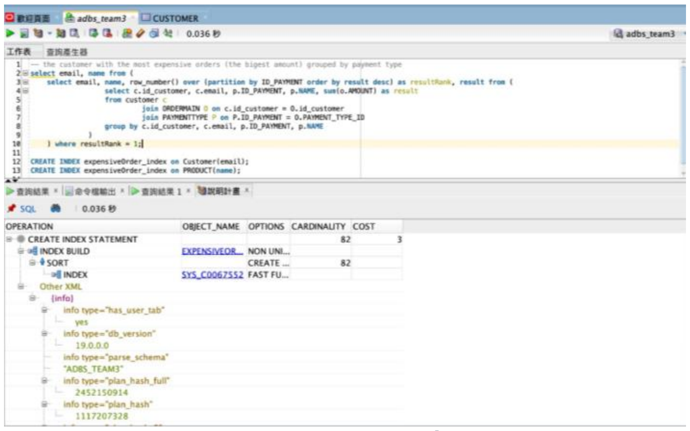
In Question 2, after creating the indexes for attribute *email* in Table *Customer* and attribute *name* in Table *Product*, can see the processing time from 0.064 second (Figure 8) decrease to 0.036 second (Figure 9). In addition, there is a remarkable point that cost from 626 (Figure 8) reduced to 3 (Figure 9). It means that the indexes are really useful.



```sql
-- the customer with the most expensive orders (the bigest amount) grouped by payment type
select email, name
from (
    select email, name, row_number() over (partition by ID_PAYMENT order by result desc) as resultRank, result from (
                select c.id_customer, c.email, p.ID_PAYMENT, p.NAME, sum(o.AMOUNT) as result
                from customer c
                        join ORDERMAIN O on c.id_customer = O.id_customer
                        join PAYMENTTYPE P on P.ID_PAYMENT = O.PAYMENT_TYPE_ID
                group by c.id_customer, c.email, p.ID_PAYMENT, p.NAME
            )
    )
where resultRank = 1;
```

```
0.064 秒
OPERATION                      OBJECT_NAME    OPTIONS            CARDINALITY  COST
SELECT STATEMENT                                                       10000   626
  VIEW                                                                 10000   626
    Filter Predicates
      RESULTRANK=1
    WINDOW                                    SORT PUSHED RANK         10000   626
      Filter Predicates
        ROW_NUMBER() OVER ( PARTITION BY P.ID_PAYMENT ORDER BY SUM(O.AMOUNT) DESC )<=1
      HASH                                    GROUP BY                 10000   626
        HASH JOIN                                                      10000    85
          Access Predicates
            P.ID_PAYMENT=O.PAYMENT_TYPE_ID
          TABLE ACCESS         PAYMENTTYPE    FULL                         3     3
          HASH JOIN                                                    10000    82
            Access Predicates
              C.ID_CUSTOMER=O.ID_CUSTOMER
            NESTED LOOPS                                               10000    82
```

Figure 8- Question 2



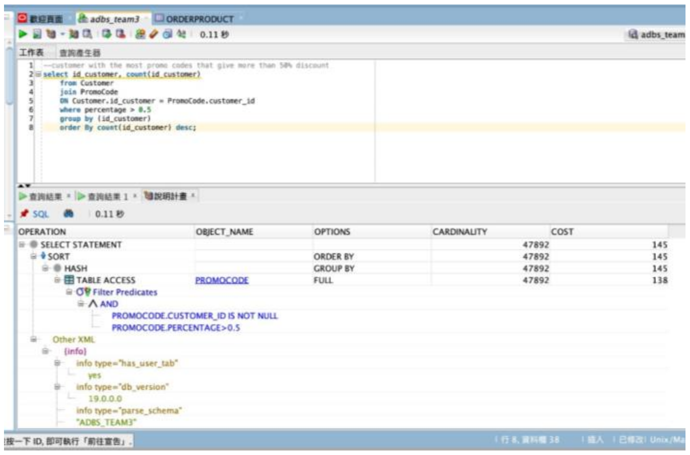
```sql
-- the customer with the most expensive orders (the bigest amount) grouped by payment type
select email, name from (
    select email, name, row_number() over (partition by ID_PAYMENT order by result desc) as resultRank, result from (
                select c.id_customer, c.email, p.ID_PAYMENT, p.NAME, sum(o.AMOUNT) as result
                from customer c
                        join ORDERMAIN O on c.id_customer = O.id_customer
                        join PAYMENTTYPE P on P.ID_PAYMENT = O.PAYMENT_TYPE_ID
                group by c.id_customer, c.email, p.ID_PAYMENT, p.NAME
            )
    ) where resultRank = 1;

CREATE INDEX expensiveOrder_index on Customer(email);
CREATE INDEX expensiveOrder_index on PRODUCT(name);
```

```
0.036 秒
OPERATION                      OBJECT_NAME    OPTIONS      CARDINALITY  COST
CREATE INDEX STATEMENT                                              82     3
  INDEX BUILD                  EXPENSIVEOR... NON UNI...
    SORT                                      CREATE ...            82
      INDEX                    SYS_C0067552   FAST FU...
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
      info type="parse_schema"
      "ADBS_TEAM3"
      info type="plan_hash_full"
        2452150914
      info type="plan_hash"
        1117207328
```

Figure 9- Question 2 Index

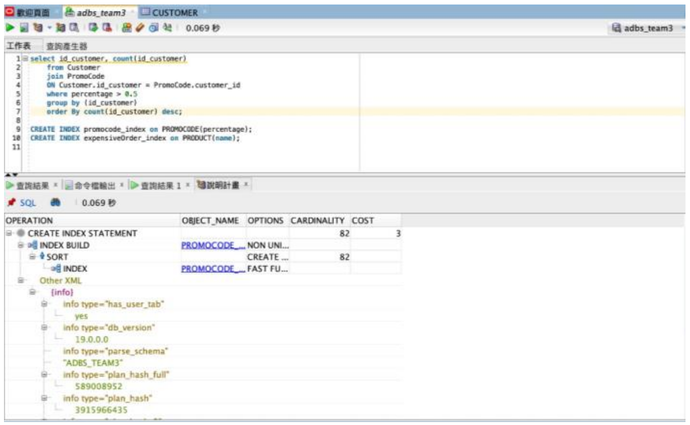
The index about attribute *percentage* in Table *promoCode* helps Question 6(Figure 10, Figure 11) and Question 7(Figure 12, Figure 13). Especially in Question 6, this index largely improves the processing time and the number of plans.



```sql
--customer with the most promo codes that give more than 50% discount
select id_customer, count(id_customer)
    from Customer
    join PromoCode
    ON Customer.id_customer = PromoCode.customer_id
    where percentage > 0.5
    group by (id_customer)
    order By count(id_customer) desc;
```

```
0.11 秒
OPERATION                      OBJECT_NAME    OPTIONS      CARDINALITY  COST
SELECT STATEMENT                                                 47892   145
  SORT                                        ORDER BY           47892   145
    HASH                                      GROUP BY           47892   145
      TABLE ACCESS             PROMOCODE      FULL               47892   138
        Filter Predicates
          AND
            PROMOCODE.CUSTOMER_ID IS NOT NULL
            PROMOCODE.PERCENTAGE>0.5
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
      info type="parse_schema"
      "ADBS_TEAM3"
```

Figure 10 - Question 6

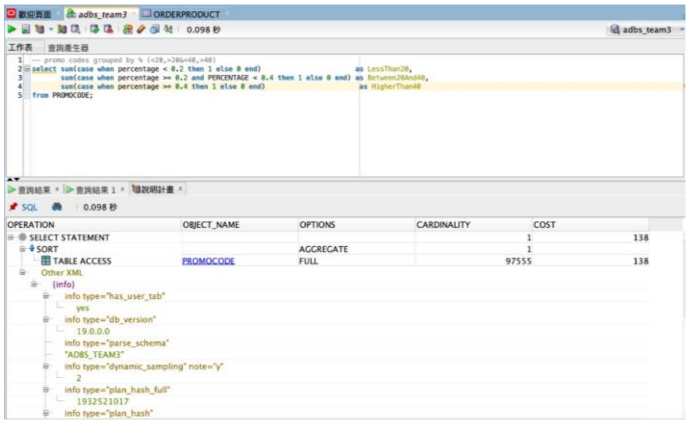


```sql
select id_customer, count(id_customer)
    from Customer
    join PromoCode
    ON Customer.id_customer = PromoCode.customer_id
    where percentage > 0.5
    group by (id_customer)
    order By count(id_customer) desc;

CREATE INDEX promocode_index on PROMOCODE(percentage);
CREATE INDEX expensiveOrder_index on PRODUCT(name);
```

```
0.069 秒
OPERATION                      OBJECT_NAME    OPTIONS      CARDINALITY  COST
CREATE INDEX STATEMENT                                              82     3
  INDEX BUILD                  PROMOCODE_...  NON UNI...
    SORT                                      CREATE ...            82
      INDEX                    PROMOCODE_...  FAST FU...
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
      info type="parse_schema"
      "ADBS_TEAM3"
      info type="plan_hash_full"
        589008952
      info type="plan_hash"
        3915966435
```

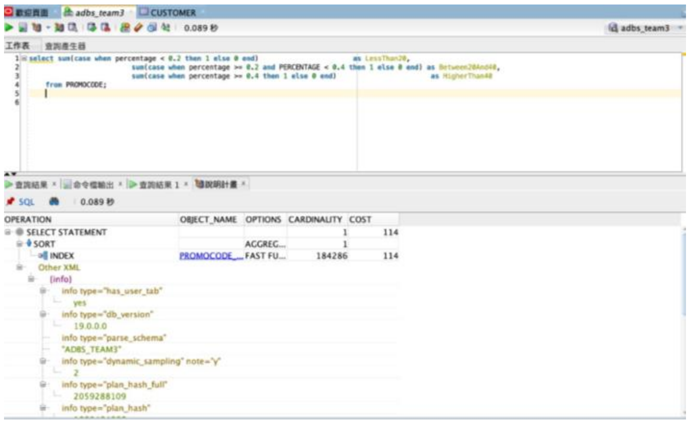
Figure 11- Question 6 Index



```sql
-- promo codes grouped by % (<20,>20&<40,>40)
select sum(case when percentage < 0.2 then 1 else 0 end)                          as LessThan20,
       sum(case when percentage >= 0.2 and PERCENTAGE < 0.4 then 1 else 0 end) as Between20And40,
       sum(case when percentage >= 0.4 then 1 else 0 end)                          as HigherThan40
from PROMOCODE;
```

```
0.098 秒
OPERATION                      OBJECT_NAME    OPTIONS      CARDINALITY  COST
SELECT STATEMENT                                                     1   138
  SORT                                        AGGREGATE              1
    TABLE ACCESS               PROMOCODE      FULL               97555   138
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
      info type="parse_schema"
      "ADBS_TEAM3"
      info type="dynamic_sampling" note="y"
        2
      info type="plan_hash_full"
        1932521017
      info type="plan_hash"
```

Figure 12 - Question 7

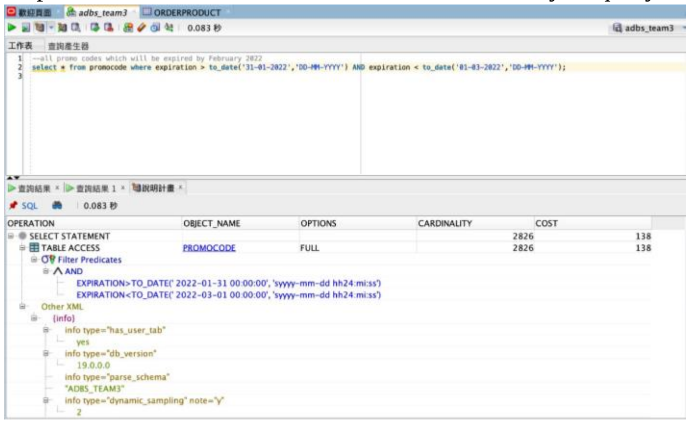


```sql
select sum(case when percentage < 0.2 then 1 else 0 end)                              as LessThan20,
       sum(case when percentage >= 0.2 and PERCENTAGE < 0.4 then 1 else 0 end) as Between20And40,
       sum(case when percentage >= 0.4 then 1 else 0 end)                           as HigherThan40
  from PROMOCODE;
```

```
0.089 秒
OPERATION                 OBJECT_NAME   OPTIONS    CARDINALITY  COST
SELECT STATEMENT                                             1   114
  SORT                                  AGGREG...            1
    INDEX                 PROMOCODE_... FAST FU...      184286   114
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
      info type="parse_schema"
      "ADBS_TEAM3"
      info type="dynamic_sampling" note="y"
        2
      info type="plan_hash_full"
        2059288109
      info type="plan_hash"
```

Figure 13 - Question 7 Index

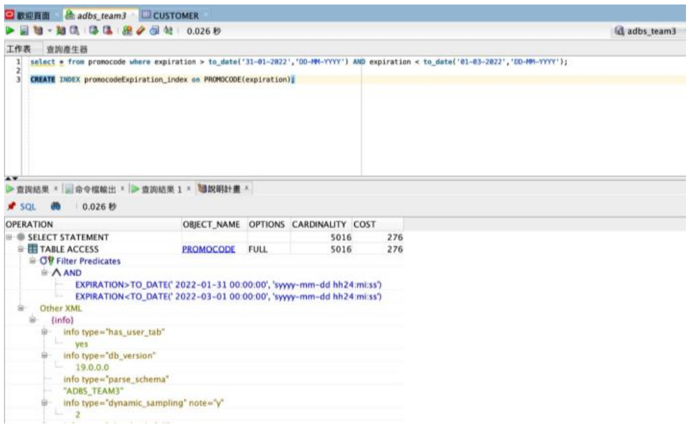
Created an index about attribute *expiration* in Table *promoCode* for Question 8. However, if comparing to Figure 14, Figure 15 and Figure 16, we can find processing time may decrease, but the cost becomes worse. Even we use parallel to make multiple processes work simultaneously. It doesn't really help. Thus, this index is not useful for the efficiency of query.



```sql
--all promo codes which will be expired by February 2022
select * from promocode where expiration > to_date('31-01-2022','DD-MM-YYYY') AND expiration < to_date('01-03-2022','DD-MM-YYYY');
```

```
0.083 秒
OPERATION                 OBJECT_NAME   OPTIONS    CARDINALITY  COST
SELECT STATEMENT                                          2826   138
  TABLE ACCESS            PROMOCODE     FULL              2826   138
    Filter Predicates
      AND
        EXPIRATION>TO_DATE(' 2022-01-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
        EXPIRATION<TO_DATE(' 2022-03-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
      info type="parse_schema"
      "ADBS_TEAM3"
      info type="dynamic_sampling" note="y"
        2
```

Figure 14 - Question 8

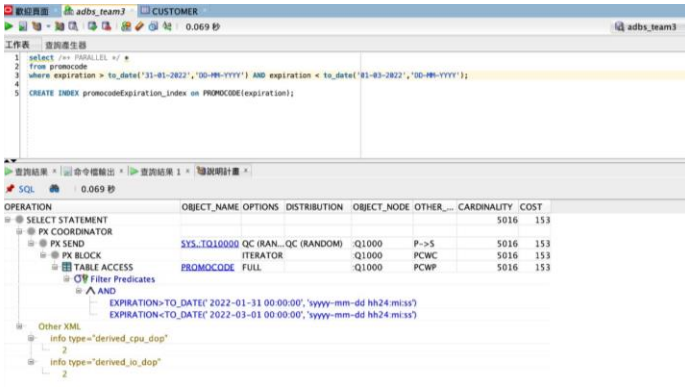


```sql
select * from promocode where expiration > to_date('31-01-2022','DD-MM-YYYY') AND expiration < to_date('01-03-2022','DD-MM-YYYY');

CREATE INDEX promocodeExpiration_index on PROMOCODE(expiration);
```

```
0.026 秒
OPERATION                 OBJECT_NAME   OPTIONS    CARDINALITY  COST
SELECT STATEMENT                                          5016   276
  TABLE ACCESS            PROMOCODE     FULL              5016   276
    Filter Predicates
      AND
        EXPIRATION>TO_DATE(' 2022-01-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
        EXPIRATION<TO_DATE(' 2022-03-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
  Other XML
    {info}
      info type="has_user_tab"
        yes
      info type="db_version"
        19.0.0.0
      info type="parse_schema"
      "ADBS_TEAM3"
      info type="dynamic_sampling" note="y"
        2
```

Figure 15 - Question 8 Index



```sql
select /*+ PARALLEL */ *
from promocode
where expiration > to_date('31-01-2022','DD-MM-YYYY') AND expiration < to_date('01-03-2022','DD-MM-YYYY');

CREATE INDEX promocodeExpiration_index on PROMOCODE(expiration);
```

```
0.069 秒
OPERATION               OBJECT_NAME   OPTIONS    DISTRIBUTION  OBJECT_NODE  OTHER_...  CARDINALITY  COST
SELECT STATEMENT                                                                              5016   153
  PX COORDINATOR
    PX SEND             SYS.:TQ10000  QC (RAN... QC (RANDOM)   :Q1000       P->S              5016   153
      PX BLOCK                        ITERATOR                 :Q1000       PCWC              5016   153
        TABLE ACCESS    PROMOCODE     FULL                     :Q1000       PCWP              5016   153
          Filter Predicates
            AND
              EXPIRATION>TO_DATE(' 2022-01-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
              EXPIRATION<TO_DATE(' 2022-03-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
  Other XML
    info type="derived_cpu_dop"
      2
    info type="derived_io_dop"
      2
```

Figure 16- Question 8 Index Parallel

C. Model review

To analyze the database, we simulate 10 questions that may be useful for business.

1) Number of orders in first quarter of years that order status is payed;
2) The customer with the most expensive order (the bigest amount) grouped by payment type;
3) Top 10 most expensive products;
4) Top 10 orders with the smallest amount to be payed
5) The order with the most products;
6) Customer with the most promo codes that give more than 50% discount;
7) Promo codes grouped by % (<20%, >20% & <40%, >40%);
8) All promo codes which will be expired by February 2022;
9) Percentage of orders where order statuse is returned;
10) Top 10 product with most orders.

About Question 1, we can understand 622 orders in the first quarter. If we also analyze other quarters, then we can find out when customers especially willing to spend money, then set a development strategy for each quartal. For Question 2, saying who cost the most and how much they paid in each payment type. Additionally, we can find customers paid by PayPal cost more than credit cards and cash showing people prefer paying directly from account than others. Question 3 shows the company is offering up to 489 euros product.

Question 4 can find the 10 smallest order amounts are from 164 euros to 999 euros. These are target customers of the company. From Question 5 shows the order has 160 products, total amount is 264 euros. However, in this ranking, since second orders all cost more than 5000 euros per order. It means the order has the most products can be an exception if a company wants to set a strategy for these customers. Also, if we combine Question 4 and Question 5, we can find that customers in this selling system may have high demand for products. In Question 6, we can see that the customer has 25 promo codes, however, he didn't use any one of them. Question 7 divided promo codes into three parts, 38,962 lower than 20%, 40,672 promo codes between 20% and 40% and 120,366 promo codes over 40%.

Question 8 shows 5,622 promo codes will expire in February. About 56% in all promo codes and the percentage of being used promo code is low. Showing customers have great loyalty to the company. They keep ordering from here even without using promo code. In Question 9, there is 25.9% of orders be returned. That means it's better that the company checks the whole supply chain, finds the problems and tries to lower the percentage of it. The last Question shows that top 10 popular products being order about 1000-1100 times and price of each product from 40 to 484 euros. This couldn't reflect the relation between customer preferences and product price, but there is a possibility between product categories and customer preferences.

III.CONCLUSION

This paper summarizes the process of creating and optimizing a database, from the initial idea and model. Observed from the formal aspect, the paper deals with the creation of the model and its realization in the database schema. The data used in the project was generated by team members. The data also contains password encryption as an important part of business logic. The database was further optimized by using horizontal partitioning and applying indexes.

The eventual continuation of work on the project would include additional improvements to the database and the creation of a user application to which this system applies.

In addition to all the above, this work is a product of the work of an international team within the Erasmus project. For this reason, it is much more than a simulation of an actual business process. During the project, we had the opportunity to understand how to process the same teaching units at different faculties. We managed to apply the theoretically acquired knowledge. A project like this encourages independent research, too. As our initial knowledge differed, we were able to learn during the process, independently, but also from each other.

Above all, the work on the project has somewhat managed to bring closer the real business environment that awaits us in the future. Therefore, the final conclusion of this project is that it has brought progress in academic, professional, but also human terms.

IV. ACKNOWLEDGMENT

Thanks to Professor Michal Kvet and Professor Srđa Bjeladinović for giving us this opportunity to challenge ourselves and leave our comfort area. During the period, we had some doubts and they gave us many useful opinions that helped us finish this project. Also, appreciate to our team members. Even if we couldn't meet each other in person, we still had a good cooperation between ourselves.