Descriptive Analysis and Visualisation Lab Report

RICHARD OGUJAWA

February 3, 2024

Description

The purpose of this report is to take a deeper look into the use of Materialized Views and Dimensional Objects in Data Warehousing, in order to understand how it's implementation and integration into OLAP systems help improve their overall performance (which will be measured by way of analysing the difference between query fetch times and the Explain Plan cost of each query).

Aims and Objectives

1. Consider the performance of a query not using a Materialised View
2. Consider the performance of the same query with a Materialised View
3. Highlight the importance of Dimensional Objects in the overall optimisation process.
4. Suggest some additional ways to further optimize query performance.

Method

1. Setup

In order be able to perform the queries, the following steps were taken:

1. Opened SQL Developer
2. Navigated to this section of the software and clicked on the highlighted button to create a new SQL script:

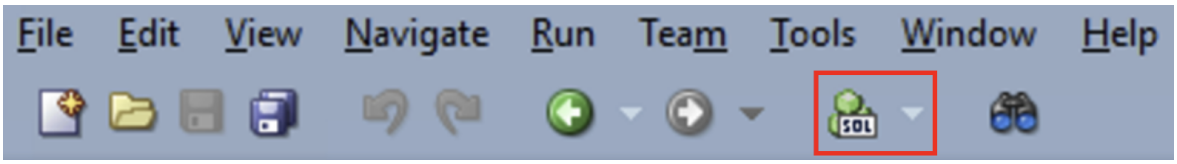


Figure 1.1: Menu Bar for SQL Developer

3. Navigated to the drop down menu on the right-hand side of the software

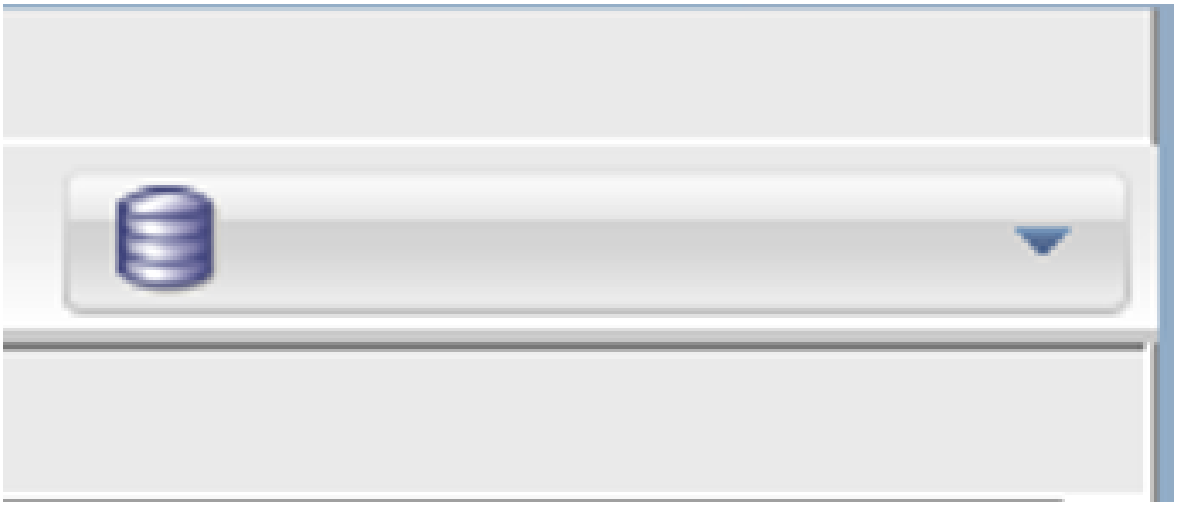


Figure 2.1: Schema Drop Down Menu

4. Selected the appropriate schema which has access to all of the necessary data to perform the queries.

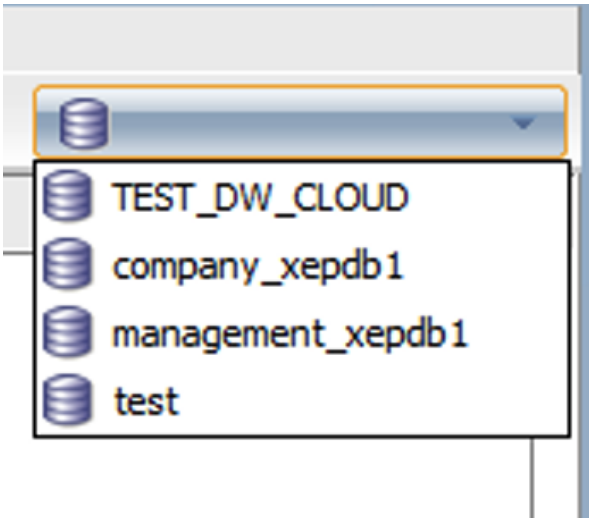


Figure 2.2: Schema Drop Down Menu Active

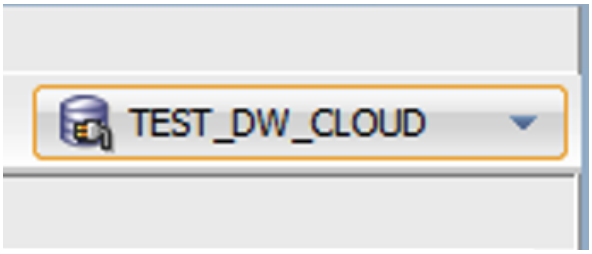


Figure 2.3: Selecting `TEST_DW_CLOUD` Schema

2. Queries

2.1 Non-Optimised Initial Query

After creating the SQL file and connecting to the appropriate schema, the next step was to perform some queries. The first query to be run was one which sought to evaluate the performance of the system without the implementation of any Materialised Views, or Dimensional Objects. The query was subsequently run using the keyboard shortcut, `CTRL + ENTER`:

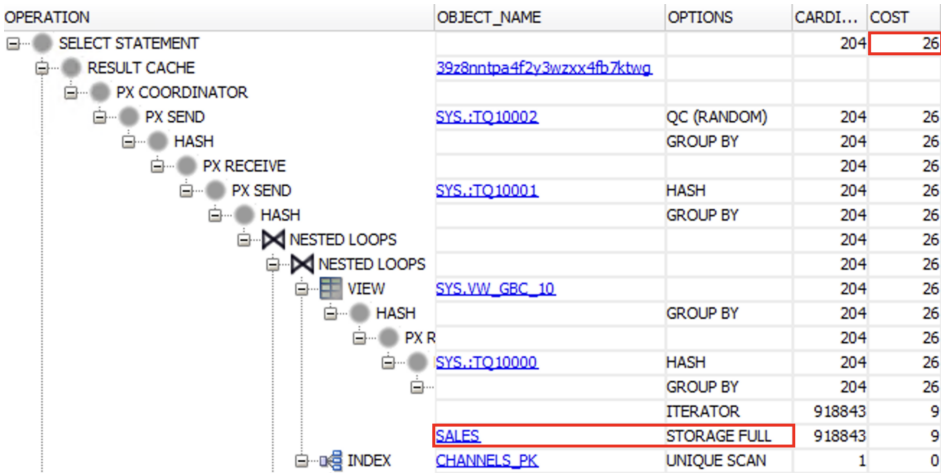
```
1  /* QUESTION 1 ~ Write a query to return sales total grouped by channel id and
2  product id. (No SQL extensions to group by required here).*/
3  SELECT c.channel_id, p.prod_id, SUM(s.amount_sold)
4  FROM sales s, channels c, products p
5  WHERE c.channel_id = s.channel_id
6  AND p.prod_id = s.prod_id
7  GROUP BY c.channel_id, p.prod_id
8  ;
```

Figure 3.1: Query to Show the Total Number of Sales Grouped by `channel_id` and `prod_id`

SQL | All Rows Fetched: 228 in 0.159 seconds

	CHANNEL_ID	PROD_ID	SUM(S.AMOUNT_SOLD)
211	2	132	98942.5
212	4	119	20330.49
213	3	142	102939.36
214	4	126	31518.02
215	2	17	2174950.74

Figure 3.2: Output Result for First Query with Fetch Time Highlighted



OPERATION	OBJECT_NAME	OPTIONS	CARDI...	COST
SELECT STATEMENT			204	26
RESULT CACHE	39z8nntpa4f2y3wzxx4fb7ktwg			
PX COORDINATOR				
PX SEND	SYS.:TQ10002	QC (RANDOM)	204	26
HASH		GROUP BY	204	26
PX RECEIVE			204	26
PX SEND	SYS.:TQ10001	HASH	204	26
HASH		GROUP BY	204	26
NESTED LOOPS			204	26
NESTED LOOPS			204	26
VIEW	SYS.VW_GBC_10		204	26
HASH		GROUP BY	204	26
PX R			204	26
	SYS.:TQ10000	HASH	204	26
		GROUP BY	204	26
		ITERATOR	918843	9
	SALES	STORAGE FULL	918843	9
INDEX	CHANNELS_PK	UNIQUE SCAN	1	0

Figure 3.3: Explain Plan with the Cost and Table Access Highlighted

The above figures showcase the query, its output and the Explain plan for the query. The pieces of information highlighted in red are the important parts of each figure with regards to the aim/goal of this report. Fig 3.2 shows the output of the query, and also displays the fetch time, which, in this particular instance was 0.095 seconds. However, it's important to note that the fetch time varies with regards to a number of factors such as whether or not the execution plan (more on this later) has been cached allowing for faster processing of said query, the quality of the network at the time of the query execution, system load (the amount of queries the data-warehouse is simultaneously processing), etc. Due to this lack of consistency, it's also a good idea to take a look at the Explain Plan for a more comprehensive understanding of the query's performance.

Oracle includes something called Oracle Optimizer which seeks to optimize queries run using on SQL Developer. As such, although a query may be expressed in a particular fashion, Oracle looks for ways in which the query can be re-written, in order to maximise the efficiency of the query's execution.

According to Oracle documentation (Oracle, 2017):

> "The Oracle Optimizer is a cost-based optimizer. The execution plan selected for a SQL statement is just one of the many alternative execution plans considered by the Optimizer. The Optimizer selects the execution plan with the lowest cost, where cost represents the estimated resource usage for that plan. The lower the

cost the more efficient the plan is expected to be. The optimizer's cost model accounts for the IO, CPU, and network resources that will be used by the query."

In order to see this optimised execution plan that has been chosen by the software, the Explain plan button is pressed, which looks like this:

Figure 3.4: Explain Plan Button Circled in Red

Going through the Explain Plan, something that's evident is the high cost to the system induced by running the query. This can be attributed to the full table scan of the Sales table in order to get the necessary data items, along with that join being performed connecting the Sales table to the Channels and Products table.

This cost, however, can be reduced, and the first measure which will be put in place in this project to help reduce this will be a Materialised View.

2.2 Materialised View

Materialized views (MVs) are queries whose resulting data output is stored in tables on the hard disk. They are extremely useful when it comes to reducing the cost of a query because when used they prevent the system from having to fetch data again and again, rather the data is fetched once and stored away for future usage in queries.

To implement the view, this SQL statement was written:

```
15  CREATE MATERIALIZED VIEW SALES_CHAN_PROD_MV
16      REFRESH FORCE ON DEMAND
17      ENABLE QUERY REWRITE
18  AS
19  SELECT c.channel_id, p.prod_id, SUM(s.amount_sold)
20  FROM sales s, channels c, products p
21  WHERE c.channel_id = s.channel_id
22  AND p.prod_id = s.prod_id
23  GROUP BY c.channel_id, p.prod_id
24  ;
```

Figure 4.1: Materialized View which stores output of Initial Query

The MV has the following settings:

- `REFRESH FORCE ON DEMAND` - This results in an MV that will be updated on demand. The way in which it will be updated is by trying to do a Fast Refresh, which means that it will keep a log of changes and update only the rows that need to be updated. If a Fast Refresh isn't possible however, then it will simply opt for a Complete Refresh which is where it disposes of all of the content of the MV and adds in the data again from scratch with the updated values.

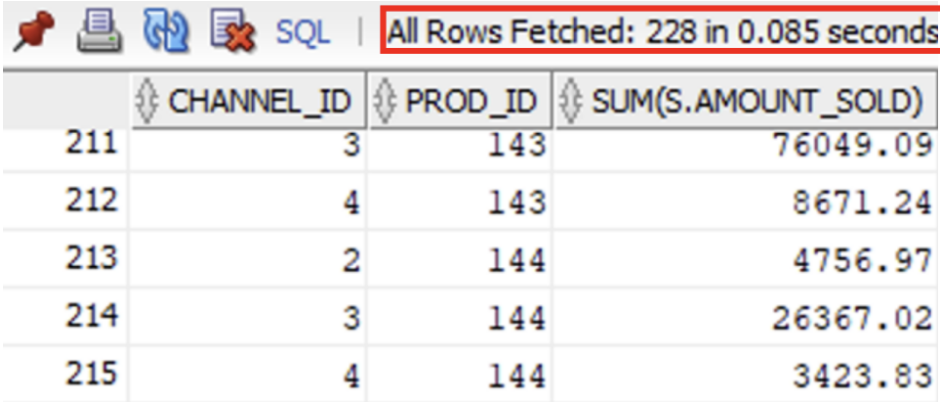
- `ENABLE QUERY REWRITE` - This command tells the system to allow query rewrite, that is to say that if the MV isn't explicitly used in a query, but can be used, that the query should be re-written to incorporate the MV.

After the query in Figure 4.1 is run to create the MV, the following message is outputted to show that the MV has been successfully created:

```
Materialized view SALES_CHAN_PROD_MV created.
```

Figure 4.2: Success Message shown when MV is created

Now if the first query from Fig 3.1 is re-run, the output looks the same, but the fetch time reduces and the Explain plan now shows that the table being accessed is the MV as opposed to the Sales Table.



	CHANNEL_ID	PROD_ID	SUM(S.AMOUNT_SOLD)
211	3	143	76049.09
212	4	143	8671.24
213	2	144	4756.97
214	3	144	26367.02
215	4	144	3423.83

Figure 4.3: Fetch Time Reduced with MV

Figure 4.4: MV Is Now Used Instead of Sales Table

2.3 Dimensional Objects

The Materialised View works for that specific query, but what about something that's not identical to the query being stored in the MV?

For example, this query, which is used to get the sales totals grouped by channel description and product name, instead of channel id and product id.

```
39 SELECT c.channel_desc AS "Channel Description", p.prod_name AS "Product Name",
40 TO_CHAR(SUM(s.amount_sold), 'L9,999,999.99') AS "Sales Total"
41 FROM sales s, channels c, products p
42 WHERE c.channel_id = s.channel_id
43 AND p.prod_id = s.prod_id
44 GROUP BY channel_desc, prod_name;
```

Figure 5.1: Query to Group Sales Totals by Channel Description and Product Name

SQL | All Rows Fetched: 225 in 0.121 seconds

	Channel Description	Product Name	Sales Total
208	Internet	O/S Documentation Set - Spanish	€36,393.02
209	Direct Sales	3 1/2" Bulk diskettes, Box of 50	€157,773.31
210	Partners	128MB Memory Card	€168,783.39
211	Partners	Home Theatre Package with DVD-Audio/...	€1,878,045.81
212	Partners	Envoy External Keyboard	€18,387.30

Figure 5.2: Output of Query from Figure 5.1

The Explain Plan shows that although there is an MV stored in the system which has already joined the sales table to the channels table and the products table, it's not being used:

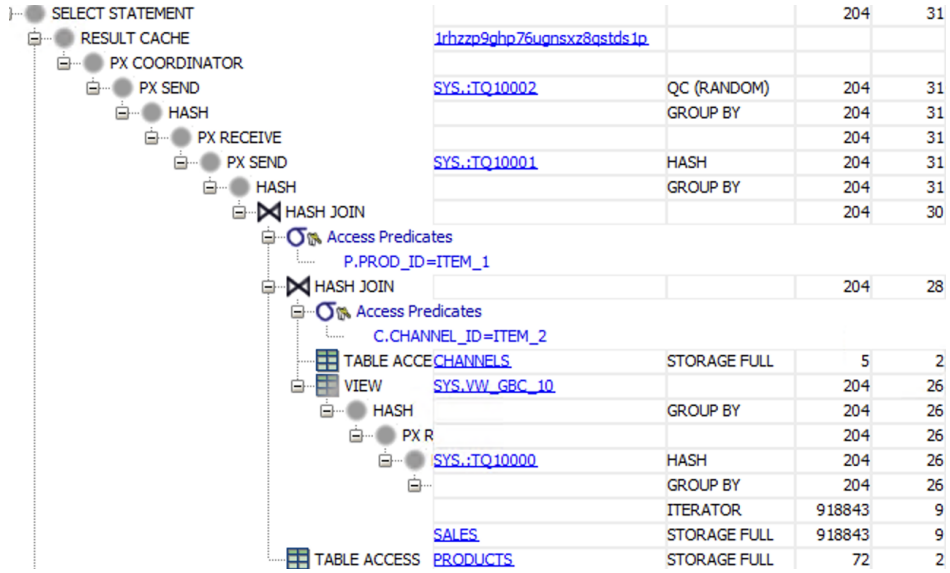


Figure 5.3: Explain Plan for Query from Figure 5.1

Why this is happening is because although the join was performed, the only information stored from the join was the sales total, product id, and the channel id, however, the system doesn't know that a relationship exists between these attributes and the product name and channel description. This is where Dimensional Objects come in. They can be used to help the system be aware of these relationships to better enhance it's ability to perform query rewrites so that in future execution plans it can use the MV even if product id and channel id are not explicitly mentioned in the query.

In order to create the Dimensional Objects the following queries were run:

```
CREATE DIMENSION PRODUCTS_DIM
  LEVEL SUBCATEGORY                    IS
    (PRODUCTS.PROD_SUBCATEGORY_ID)
  LEVEL CATEGORY                       IS
    (PRODUCTS.PROD_CATEGORY_ID)
  LEVEL PROD_TOTAL                     IS
    (PRODUCTS.PROD_TOTAL_ID)
  LEVEL PRODUCT                        IS
    (PRODUCTS.PROD_ID)
  HIERARCHY PROD_ROLLUP
    (PRODUCT                           CHILD OF
     SUBCATEGORY                       CHILD OF
     CATEGORY                          CHILD OF
     PROD_TOTAL)
  ATTRIBUTE CATEGORY DETERMINES
    (PRODUCTS.PROD_CATEGORY,
     PRODUCTS.PROD_CATEGORY_DESC)
  ATTRIBUTE PRODUCT DETERMINES
    (PRODUCTS.PROD_NAME,
     PRODUCTS.PROD_DESC,
     PRODUCTS.PROD_WEIGHT_CLASS,
     PRODUCTS.PROD_UNIT_OF_MEASURE,
     PRODUCTS.PROD_PACK_SIZE,
     PRODUCTS.PROD_STATUS,
     PRODUCTS.PROD_LIST_PRICE,
     PRODUCTS.PROD_MIN_PRICE)
  ATTRIBUTE PROD_TOTAL DETERMINES
    (PRODUCTS.PROD_TOTAL)
  ATTRIBUTE SUBCATEGORY DETERMINES
    (PRODUCTS.PROD_SUBCATEGORY,
     PRODUCTS.PROD_SUBCATEGORY_DESC);
```

Figure 6.1: Products' Dimension

```
183 CREATE DIMENSION CHANNELS_DIM
184 --Set up the levels
185   LEVEL CHANNEL IS (CHANNELS.CHANNEL_ID)
186   LEVEL CLASS IS (CHANNELS.CHANNEL_CLASS_ID)
187   LEVEL TOTAL IS (CHANNELS.CHANNEL_TOTAL_ID)
188 --Set up the hierarchy for the roll-up
189   HIERARCHY CHANNEL_ROLLUP
190     (CHANNEL CHILD OF
191     CLASS CHILD OF
192     TOTAL)
193 --Set up the attributes that each level relates to
194   ATTRIBUTE CHANNEL DETERMINES (CHANNELS.CHANNEL_DESC)
195   ATTRIBUTE CLASS DETERMINES(CHANNELS.CHANNEL_CLASS)
196   ATTRIBUTE TOTAL DETERMINES(CHANNELS.CHANNEL_TOTAL);
```

Figure 6.2: Channels' Dimension

The creation of the Dimensional Objects following the same pattern:

- The levels are defined, and placed within a hierarchy to establish the top-down relationship that exists within the dimension.
- Then the other attributes which are to be associated with each level are also incorporated into the Dimensional Object using the `'ATTRIBUTE <level> DETERMINES <attributes>'` command.

After running the commands for the two Dimensional Objects, the following message is outputted.

```
Dimension PRODUCTS_DIM created.

Dimension CHANNELS_DIM created.
```

Figure 6.3: Dimensions Created

When the query is re-run, and the Explain Plan is displayed again, what's clear is that the cost has been significantly reduced from 31 to 7, and the MV is now also being used as part of the query, showing that the query was rewritten to incorporate it before being executed.

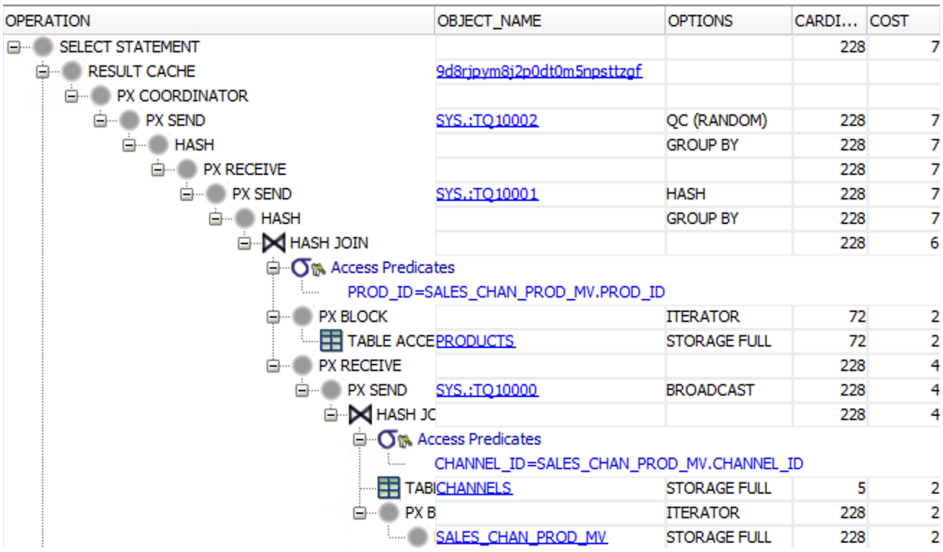


Figure 6.4: Explain Plan with MV and Dimensional Object

2.5 Functions for Grouping

Optimising query performance is more than MV's and Dimensional Objects however. Some other useful tools include indexing, database sharding and simply just writing better queries. SQL has a lot of functions that have been created which abstract away a lot of the complexity, resulting in simpler query constructions and therefore faster querying performance. An example of such can be found in the functions to help with grouping and aggregating query results:

- `ROLLUP` - This is a function which is used to aggregate the results at each of the levels defined in the hierarchy of a Dimensional Object. It's also important to note two things in the query: (1) The Grouping function is also used to help identify rows that are simply subtotals and totals (2) this information is then used in the Decode function to replace the null values present at the subtotal and total rows with useful descriptive text.

```
SELECT
    DECODE(GROUPING(c.channel_desc), 1, 'All Channels', c.channel_desc) CHANNEL,
    DECODE(GROUPING(p.prod_category), 1, 'Category All Categories', CONCAT('Category ', p.prod_category)) as CATEGORY,
    DECODE(GROUPING(sh.country_name), 1, 'Totals in France and Italy', sh.country_name) AS COUNTRY,
    TO_CHAR(SUM(amount_sold), 'L9,999,999.99') AS "Total Sales"
FROM sales s, channels c, products p, customers cu, shcountries sh
WHERE c.channel_id = s.channel_id
AND p.prod_id = s.prod_id
AND s.cust_id = cu.cust_id
AND cu.country_id = sh.country_id

AND prod_category NOT IN ('Peripherals and Accessories', 'Hardware', 'Photo')
AND sh.country_name IN ('France', 'Italy')
GROUP BY ROLLUP(c.channel_desc, p.prod_category, sh.country_name)
ORDER BY c.channel_desc
;
```

Figure 7.1: ROLLUP Example Query

SQL | All Rows Fetched: 29 in 0.077 seconds

	CHANNEL	CATEGORY	COUNTRY	Total Sales
1	Direct Sales	Category Electronics	France	€375,510.96
2	Direct Sales	Category Electronics	Italy	€439,978.92
3	Direct Sales	Category Electronics	Totals in France and Italy	€815,489.88
4	Direct Sales	Category Software/Other	France	€338,738.06
5	Direct Sales	Category Software/Other	Italy	€404,966.90
6	Direct Sales	Category Software/Other	Totals in France and Italy	€743,704.96
7	Direct Sales	Category All Categories	Totals in France and Italy	€1,559,194.84
8	Internet	Category Electronics	France	€76,243.30
9	Internet	Category Electronics	Italy	€120,493.13
10	Internet	Category Electronics	Totals in France and Italy	€196,736.43
11	Internet	Category Software/Other	France	€49,446.28
12	Internet	Category Software/Other	Italy	€68,513.07

Figure 7.2: Output from ROLLUP Query

- `PARTIAL ROLLUP` - This works in an almost idential way to ROLLUP, the only difference being that ROLLUP occurs when all of the non-aggregate columns in the SELECT statement are rolled-up into their subtotals and totals. PARTIAL ROLLUP, on the other hand, occurs when at least one or more of the non-aggregate columns in the SELECT statement are not included in the roll up.

```
SELECT
    DECODE(GROUPING(c.channel_desc), 1, 'All Channels', c.channel_desc) CHANNEL,
    DECODE(GROUPING(p.prod_category), 1, 'Category All Categories', CONCAT('Category ', p.prod_category)) as CATEGORY,
    DECODE(GROUPING(sh.country_name), 1, 'Totals in France and Italy', sh.country_name) AS COUNTRY,
    TO_CHAR(SUM(amount_sold), 'L9,999,999.99') AS "Total Sales"
FROM sales s, channels c, products p, customers cu, shcountries sh
WHERE c.channel_id = s.channel_id
AND p.prod_id = s.prod_id
AND s.cust_id = cu.cust_id
AND cu.country_id = sh.country_id

AND prod_category NOT IN ('Peripherals and Accessories', 'Hardware', 'Photo')
AND sh.country_name IN ('France', 'Italy')
/*Syntax for Partial Rollup - everything has to be included in the group by statement that's
not an aggregator, so group the column names normally and then rollup the ones you want to.
Hence 'Partial' Rollup.*/
/*Error Report: Initially I tried to just use PARTIAL ROLLUP and it didnt' work because it
was saying that the SQL command didn't end properly. The I tried to just ROLLUP certain columnns
and then I was told that it wasn't a proper GROUP BY expression because columns were ommitted.*/
GROUP BY sh.country_name, ROLLUP(c.channel_desc, p.prod_category)
ORDER BY c.channel_desc, p.prod_category;
```

Figure 7.3: PARTIAL ROLLUP Example Query

	CHANNEL	CATEGORY	COUNTRY	Total Sales
1	Direct Sales	Category Electronics	France	€375,510.96
2	Direct Sales	Category Electronics	Italy	€439,978.92
3	Direct Sales	Category Software/Other	France	€338,738.06
4	Direct Sales	Category Software/Other	Italy	€404,966.90
5	Direct Sales	Category All Categories	Italy	€844,945.82

Figure 7.4: Output from PARTIAL ROLLUP Query

- `CUBE` - This does the same thing as ROLLUP, however, it also provides subtotals for all the combinations of the grouped columns in the GROUP BY clause.

```
263 -- CUBE
264 SELECT
265     DECODE(GROUPING(c.channel_desc), 1, 'All Channels', c.channel_desc) CHANNEL,
266     DECODE(GROUPING(p.prod_category), 1, 'Category All Categories', CONCAT('Category ', p.prod_category)) as CATEGORY,
267     DECODE(GROUPING(sh.country_name), 1, 'Totals in France and Italy', sh.country_name) AS COUNTRY,
268     TO_CHAR(SUM(amount_sold), 'L9,999,999.99') AS "Total Sales"
269 FROM sales s, channels c, products p, customers cu, shcountries sh
270 WHERE c.channel_id = s.channel_id
271 AND p.prod_id = s.prod_id
272 AND s.cust_id = cu.cust_id
273 AND cu.country_id = sh.country_id
274
275 AND prod_category NOT IN ('Peripherals and Accessories', 'Hardware', 'Photo')
276 AND sh.country_name IN ('France', 'Italy')
277 -- Syntax for CUBE
278 GROUP BY CUBE(c.channel_desc, p.prod_category, sh.country_name)
279 ORDER BY c.channel_desc, p.prod_category;
```

Figure 7.5: CUBE Example Query

	CHANNEL	CATEGORY	COUNTRY	Total Sales
1	Direct Sales	Category Electronics	France	€375,510.96
2	Direct Sales	Category Electronics	Italy	€439,978.92
3	Direct Sales	Category Electronics	Totals in France and Italy	€815,489.88
4	Direct Sales	Category Software/Other	France	€338,738.06
5	Direct Sales	Category Software/Other	Italy	€404,966.90

Figure 7.6: Output from CUBE Query

- `GROUPING SETS` - This is similar to CUBE in that it provides subtotals for combinations of columns in the GROUP BY clause, but it allows you to dictate which columns you want to group together and therefore see subtotals for.

```
281 -- GROUPING SETS
282 SELECT
283     DECODE(GROUPING(c.channel_desc), 1, 'All Channels', c.channel_desc) CHANNEL,
284     DECODE(GROUPING(p.prod_category), 1, 'Category All Categories', CONCAT('Category ', p.prod_category)) as CATEGORY,
285     DECODE(GROUPING(sh.country_name), 1, 'Totals in France and Italy', sh.country_name) AS COUNTRY,
286     TO_CHAR(SUM(amount_sold), 'L9,999,999.99') AS "Total Sales"
287 FROM sales s, channels c, products p, customers cu, shcountries sh
288 WHERE c.channel_id = s.channel_id
289 AND p.prod_id = s.prod_id
290 AND s.cust_id = cu.cust_id
291 AND cu.country_id = sh.country_id
292
293 AND prod_category NOT IN ('Peripherals and Accessories', 'Hardware', 'Photo')
294 AND sh.country_name IN ('France', 'Italy')
295 -- Syntax for CUBE
296 GROUP BY GROUPING SETS (
297     (c.channel_desc, p.prod_category),
298     (c.channel_desc, sh.country_name),
299     () -- get the total
300 );
```

Figure 7.7: CUBE Example Query

SQL | All Rows Fetched: 17 in 0.083 seconds

	CHANNEL	CATEGORY	COUNTRY	Total Sales
1	Partners	Category All Categories	France	€244,865.32
2	Direct Sales	Category All Categories	France	€714,249.02
3	Partners	Category All Categories	Italy	€352,297.12
4	Tele Sales	Category All Categories	France	€29.97
5	Tele Sales	Category Electronics	Totals in France and Italy	€55.93

Figure 7.8: Output from CUBE Query

Conclusion

When it comes to writing SQL queries optimisation needs to be at the forefront of the developers mind due to the reduced fetch time and cost to the system. This report details a number of ways in which this can be done. Firstly by showing a non-optimized query, then introducing a Materialised View in subsection 2.3, which significantly improved the performance of the query, evident in its cost going from 26 to 2. Then another non-optimised query was introduced in sub-section 2.4 where it was later optimised by using a Dimensional Object to allow it to use the Materialised View created in subsection 2.3. Once again this resulted in a reduction in cost, this time from 31 to 7. Finally some additional techniques were mentioned in subsection 2.5 which also play a part in query optimisation, namely indexing, database sharding and constructing better queries using functions available to us to abstract away some of the potential complexities of our queries.

References

1. Oracle (2017). The Oracle Optimizer Explain the Explain Plan [online] Available at: https://www.oracle.com/docs/tech/database/technical-brief-explain-the-explain-plan-052011.pdf [Accessed 24 Oct. 2023].