COMP810 Data Warehousing and Big Data Assessment 2 Data Warehousing Project

Building and Analysing a DW for NatureFresh Stores in NZ

Stone Fang (Student ID: 19049045)

1 Project Overview

The goal of this project is to create a Data Warehouse (DW) for the sales analysis of NatureFresh, one of the largest fresh food market chains in New Zealand. Analysis of sales and customer shopping behaviours can give NatureFresh in-depth insight of the market, so they can improve their selling strategies accordingly.

The original available data are customer transactions and master data of product information. The transaction data contains customer shopping records, including who (customer) bought what (product) at when (date), where(store) and how many was bought (quantity). The product data contains information for each product, including supplier and price.

However, the format of original data doesn't fit the requirement of OLAP, so first we need transform the data into other formats for better querying.

The major content of this project contains:

- Design and implementation of the star-schema for sales DW, that is, fact and dimension tables.
- Filling DW by ETL process. Specifically, do Index Nested Loop Join (INLJ) on transactions and master data, transform and load data into fact & dimension tables.
- Analysis on DW by SQL queries.

All the operations above are implemented in SQL & PL/SQl.

2 Schema for DW

According to the fields in original data, the DW will consist of one fact table *Sales* and five dimension tables *Product*, *Supplier*, *Customer*, *Store*, and *Date*, as shown in figure 1. The SQL code to create all tables are in file *createDW.sql*.

2.1 Fact Table

Apparently the fact table should have foreign keys corresponding to all five dimension tables, and the quantity of item sold. There are two decisions have been make for primary key and amount of money in sales.

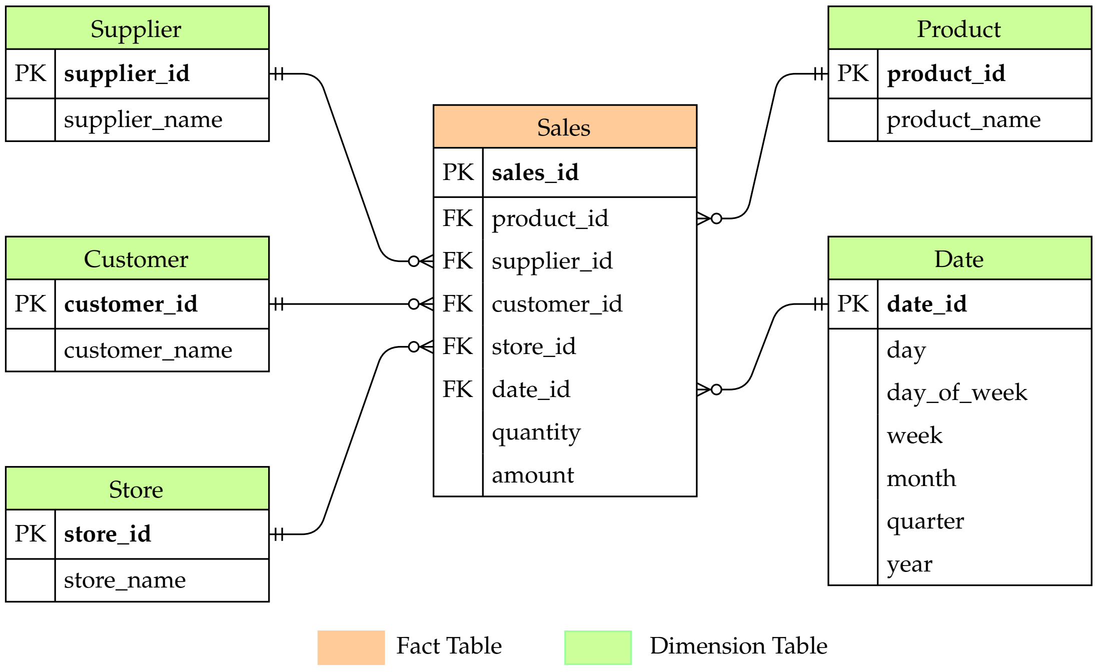


Figure 1: Star Schema of NatureFresh Sales

Primary key of fact table can be a combination of all foreign keys. However, there could be a concern to have more than one transactions for the same values on all five dimensions. A quick analysis shows that such situation does exists, though the possibility is low. In other words, a customer may buy one product multiple times at one store in one day. There are two options to solve this problem. One is summing up the quantities of multiple transactions, resulting in only one record for the same combination of dimension values. The other is keep multiple transactions while use a separated ID field as the primary key of *Sales* fact table. In this project, the latter solution is preferred because this approach can keep the original granularity of transactions, thus contains more information. Also, the possibility of multiple transactions for one combination of dimensions is low, so there would not be significant overhead in terms of memory and storage.

Price/Amount is another concerning field. In the original data, *price* is stored in master data table as a property of product, so it is natural to make it an attribute of product dimension. However, this design has a shortcoming when price changes as it always does. If the price of a product changes, we can't simply modify the value in *Product* dimension table otherwise the result on sales before that change will be incorrect. Therefore, in this project price information is kept in *Sales* table. Since the amount of money in sales is a more frequent used number, we add to fact table an *amount* filed which is calculated by $quantity \times price$. In section 5.1 further discussions will be provided on this issue.

2.2 Dimensions

Details of dimension tables can be referred to figure 1. Most dimensions are as simple as "ID"+"name", while the *Date* dimension is relatively complicated. First of all, unlike other dimensions, there is no existing ID for *Date*. In this project, a string in format of "YYYYMMDD" is chosen as the ID for *Date*, rather than an auto-incremental column. The advantage is such ID is more readable and intuitive, and thus more convenient for partitioning if required in the future. On the other hand, it would need more storage space, which, however, is not a big issue providing the cost storage is quite low nowadays.

Second, *Date* dimension contains more information other than names. In this project, common properties are calculated, including *year*, *quarter*, *month*, *week*, *day*, *day_of_week*. In fact, it can be extended to more fields such as *is_public_holiday*, if some analysis on holiday is in demand.

In Oracle SQL Developer, index will be created automatically on primary key column. However, we rarely retrieve data only by primary keys because primary key is usually an ID which is not readable to humans. There for, indexes will be created on all levels of attributes in dimension tables for fast retrieval.

3 INLJ Algorithm

Index Nested Loop Join (INLJ) is a table joining algorithm that can be used for stream data joining. Nested Loop Join takes an outer loop and an inner loop, each on one table, and output the rows that matches the conditions, so the time complexity is $O(NM)$ where N and M are the number of rows of two tables. However, INLJ only keep the outer loop and replace the inner loop with an index-based look-up, thus greatly reduce the time complexity. For example, if the index is implemented by a B-tree, then complexity of look-up is $O(\log M)$ instead of linear which is the case of the inner loop.

This algorithm is implemented in PL/SQL. First, a bulk (50 rows in this project) of transactions are read into memory. Then all rows in the bulk are read sequentially, and retrieve the information for current row from master data by *product_id* . Then all properties corresponding to current row are transformed to fit the star schema and then load into the fact and dimension tables. Please refer to file *INLJ.sql* for the complete implementation.

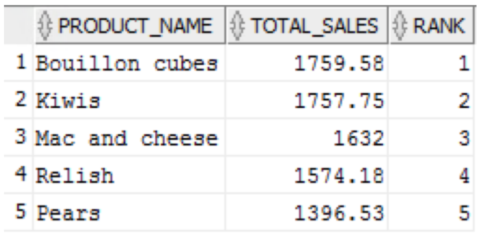
4 OLAP Queries Results

This section summarises the methods and results of required analysis. The SQL statements of these queries are referred to file *queriesDW.sql*.

Question 1

Question*: Determine the top 5 products in Dec 2019 in terms of total sales*

The basic idea is to calculate the sales per product in Dec 2019, then rank the sales in descending order, and finally get first 5 ranks. The output of query is shown as figure 2.



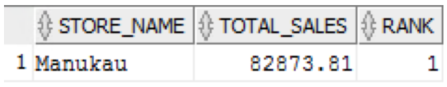
	PRODUCT_NAME	TOTAL_SALES	RANK
1	Bouillon cubes	1759.58	1
2	Kiwis	1757.75	2
3	Mac and cheese	1632	3
4	Relish	1574.18	4
5	Pears	1396.53	5

Figure 2: Query 1 Output

Question 2

> ***Question**: Determine which store produced highest sales in the whole year?*

The method is similar to previous question. The output of query is shown as figure 3.



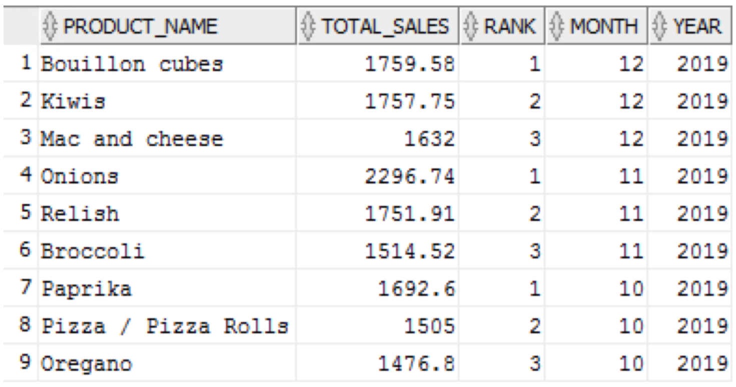
	STORE_NAME	TOTAL_SALES	RANK
1	Manukau	82873.81	1

Figure 3: Query 2 Output

Question 3

> ***Question**: Determine the top 3 products for a month (say, Dec 2019), and for the 2 months before that, in terms of total sales.*

The basic idea is to create a PL/SQL function that returns a collection mimicking a table. To do this, first we need declare the types for that collection as the return type of the function, and then use a cursor inside the function to retrieve the data of one month. Since the cursor is parameterised, it can be reused to query the data for any month. With this approach we can do the analysis for any consecutive months. The output of query is shown as figure 4.



	PRODUCT_NAME	TOTAL_SALES	RANK	MONTH	YEAR
1	Bouillon cubes	1759.58	1	12	2019
2	Kiwis	1757.75	2	12	2019
3	Mac and cheese	1632	3	12	2019
4	Onions	2296.74	1	11	2019
5	Relish	1751.91	2	11	2019
6	Broccoli	1514.52	3	11	2019
7	Paprika	1692.6	1	10	2019
8	Pizza / Pizza Rolls	1505	2	10	2019
9	Oregano	1476.8	3	10	2019

Figure 4: Query 3 Output

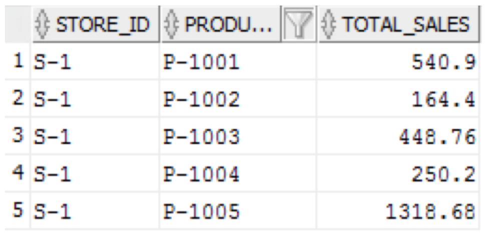
Question 4

> ***Question**: Create a materialised view called "STOREANALYSIS" that presents the product-wise sales analysis for each store. The results should be ordered by StoreID and then ProductID.*

The query is a creation statement so the output is insignificant. A sample of rows from the materialised view is shown in figure 5.

Question 5

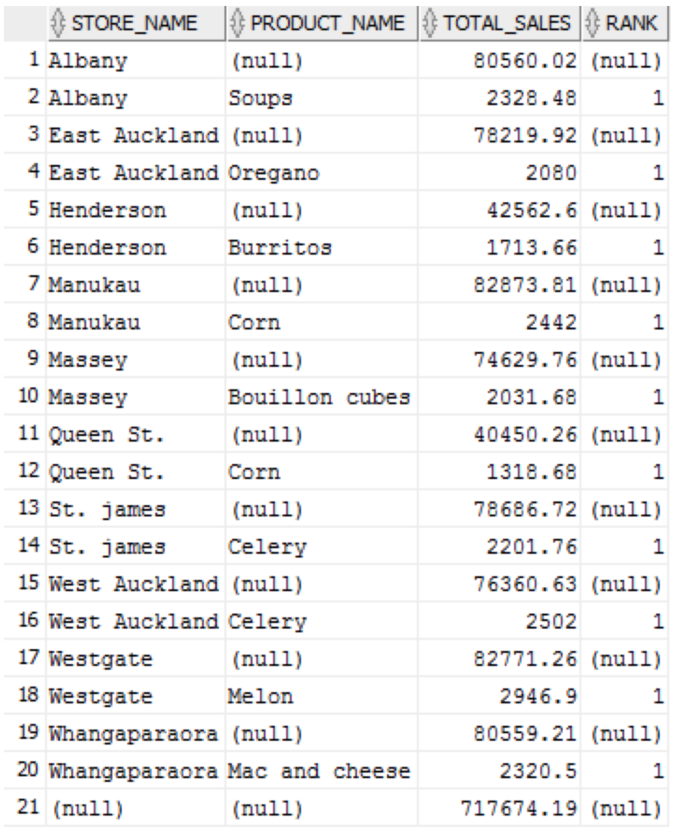
> ***Question**: Think about what information can be retrieved from the materialised view created in Q4 using ROLLUP or CUBE concepts and provide some useful information of your choice for management.*



	STORE_ID	PRODU...	TOTAL_SALES
1	S-1	P-1001	540.9
2	S-1	P-1002	164.4
3	S-1	P-1003	448.76
4	S-1	P-1004	250.2
5	S-1	P-1005	1318.68

Figure 5: Query 4 Output

To answer this question, a query will be created to calculate the overall and store-wise sales, as well as the product with highest sales in each store. `ROLLUP` is used to calculate the summation and product-wised sales simultaneously. The output of query is shown as figure 6.



	STORE_NAME	PRODUCT_NAME	TOTAL_SALES	RANK
1	Albany	(null)	80560.02	(null)
2	Albany	Soups	2328.48	1
3	East Auckland	(null)	78219.92	(null)
4	East Auckland	Oregano	2080	1
5	Henderson	(null)	42562.6	(null)
6	Henderson	Burritos	1713.66	1
7	Manukau	(null)	82873.81	(null)
8	Manukau	Corn	2442	1
9	Massey	(null)	74629.76	(null)
10	Massey	Bouillon cubes	2031.68	1
11	Queen St.	(null)	40450.26	(null)
12	Queen St.	Corn	1318.68	1
13	St. james	(null)	78686.72	(null)
14	St. james	Celery	2201.76	1
15	West Auckland	(null)	76360.63	(null)
16	West Auckland	Celery	2502	1
17	Westgate	(null)	82771.26	(null)
18	Westgate	Melon	2946.9	1
19	Whangaparaora	(null)	80559.21	(null)
20	Whangaparaora	Mac and cheese	2320.5	1
21	(null)	(null)	717674.19	(null)

Figure 6: Query 5 Output

5 Discussion

5.1 Price Attribute

In this design, the price information is beared in the *amount* field of fact table because of the fact that prices of products are subject to change. However, changes of prices are way less frequent than transactions, so there will be much redundant storage for price information. An alternative design is to add an extra dimension *SellingItem* which is simply a combination of *Product* and *price*. When price changes, a new "selling item" will be created with the same *product_id* and the new *price*. This method is show in figure 7.

The benefit is reducing the required storage for price by normalisation. However, this alternative design ends up with an architecture other than star-schema.

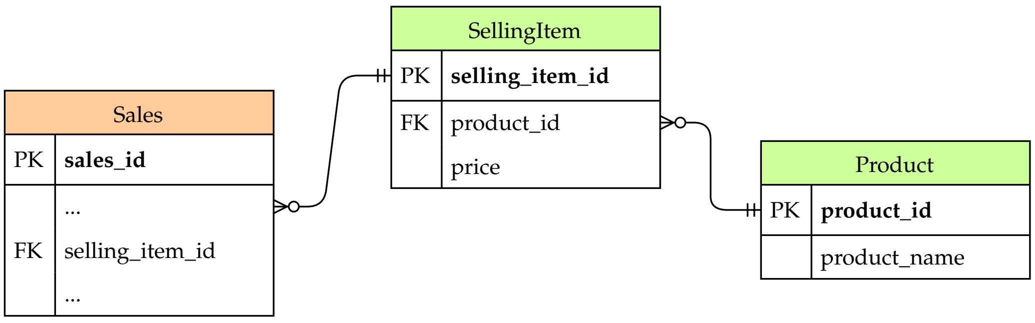


Figure 7: Alternative Design for Price Attribute

6 Summary of Learning Outcomes

It is a great way to learn from developing a complete DW project. The main outcomes learned from this project includes DW design, ETL process, DW queries, and Oracle PL/SQL and SQL features.

6.1 DW Development Lifecycle

The first and foremost thing I learned is how to develop a DW through the whole lifecycle. It involves requirement analysis, schema design, data ETL, and querying. We need to divide this complex task into different stages, and also be able to combine them together back into a complete and running DW project.

6.2 DW Schema

The DW is usually modelled as data cube which consists of dimensions and measurements. In terms of physical model, dimensions are mapped to dimension tables and measurements are mapped to fact tables. There are two major schema for DW design, star-schema and snowflake schema. In this project we use star-schema, which is more efficient for querying because it has less table joints for a query. However, the disadvantage is the lack of normalisation. Most dimensions are simple, containing only one attributed of "name". However, the *Date* dimension is trickier. Decisions have been made on the choice of primary key of *date*, and what attributes should be included in this dimension. Furthermore, constraints are applied to all fields to make all values valid. For example, *day_of_week* field is restricted to numbers between 1 and 7, so an invalid value such as 8 can never been inserted into the table.

6.3 INLJ & ETL

INLJ is suitable for joining stream data with batch data in near real-time. The key idea of INLJ of two tables is looping on the first table and looking up on the second table through an index. Apparently, index cannot be build on the stream data, so we can only create index on the batch data table and loop on the stream data. It is an effective way to avoid exhaustive search on the batch data table.

To implement INLJ in PL/SQL, "cursor" is an important tool. A cursor fetches data from the output of a query, and can be traversed by a for-loop. Cursors can fetch data row by row or in bulk. The difference between the two manner is performance. Each time when a cursor fetches data, there will be context switching between PL/SQL engine and SQL engine, which incurs overhead decreasing the overall performance. Therefore, fetching the data bulk by bulk (e.g. 50 rows at one time) can significantly increase the execution performance. To access the data from a cursor we need to declare variables to hold that data. We can either declare a single variable in type of the whole row or multiple variables each one for a specific field.

Before inserting values into dimension tables, it is necessary to check whether that value already exists in the table or not. There are multiple ways to check the existance of one row before insertion. In this project a `WHERE NOT EXISTS` clause is used along with `INSERT INTO ... SELECT ...` statement for this purpose, but the trick here is the use of a dummy table `dual`. The reason is as follows. The `INSERT INTO ... SELECT` statement will select rows from a table and insert into the target table, but here the data to insert actually comes from PL/SQL variables. Therefore, the dummy table `dual` is put in the `FROM` clause, but the columns in `SELECT` clause are actually the variables. With this approach we can make use of `INSERT INTO ... SELECT` statement to check the existance of a row before inserting it.

6.4 DW Query & Analysis

Analysis on DW should be implemented by SQL queries. It usually involves table joining so the query would be a bit complicated even for a simple analysis.

Top K retrieval is a type of common analysis, which finds the most or least measurements with corresponding dimension values. For example, it's useful for decision-making in business operation to know the products with K highest or lowest sales. This can be done by either `ORDER BY` or `RANK()` operations in Oracle SQL Developer. More than that, the analytics are usually done with roll-up/drill-down/slice/dice operations. For instance, it's common to aggregate the data of each day to month level (roll-up), or retrieve data of a certain month (slice) or a month range (dice). For this purpose, the `WHERE` and `GROUP BY` clause are used, sometimes with `PARTITION BY` operation for advanced query.

Dynamic query is an important way to improve flexibility and query reuse. Dynamic query is constructed in runtime with parameters substituted by real values assigned to them. Next time if we want to do similar analysis, we just need to change the parameter rather than manually editing the query statement. To implement a dynamic query is basically to create a PL/SQL table function with the query wrapped inside it. It's called "table function" because it returns collections of objects that minic tables. However, the details are somewhat complicated. What I learned from developing a dynamic query includes:

- Create types of expected table for return type declaration in function by `CREATE TYPE`.
- In side the function, create a parameterised cursor for data fetching. The parameters are about the target month, so the query can be used to select data of any specified month.
- Use `FOR LOOP` and `IF THEN ELSE` to iteractively fetch data from database, and convert the retrieved data into the type matching the function declaration.
- Use `PIPELINED` and `PIPE ROW` together to return the fetched records in an convenient way. Without this approach we would have to create a local collection, append records to it and return it after all data has been retrieved, but with `PIPELINED` we can simply "pipe" out each record immediately after it's retrieved without the needs for local collection.