

Summary Reports

Average monthly sales, in terms of quantity sold, per product line

Using Excel, the data in the original .csv files are loaded into a new worksheet with the “From Text / CSV”. This converts the data from a simple text / semicolon-separated state into a form that excel can perform operations on.

Under queries and connections, the newly added tables should be available; in this case, the goProducts and goDailySales can now be merged through the “product number” field. From here, the specific relevant columns can be selected to be merged (Product Line, Product Brand, Unit Cost, and Unit Price for these reports).

Now with all the relevant columns in the same sheet, a Pivot Table can be created (Insert -> Pivot Table). All the relevant fields can be specified for the rows, columns, and values. For the first query, Product Line is put under the pivot table’s “Rows” field.

This definition of average monthly sales from (SHIPHYPE Fullfillment, 2022) will be used: “The total number of sales made in a period divided by the total number of months. Search for Something Else”. To do this, the total quantity of sales as well as the total number of months will be needed. Total sales can be calculated as a summation of the quantity field per product line, and the total number of months is taken by getting the difference between the maximum and minimum dates (i.e. time between the earliest sale of a product line and the latest).

For report 1A, product brand can simply be added as another entry in the “Rows” field of the pivot table, and the appropriate values will also be calculated. For report 1B, the SUMIFS function can be used to selectively aggregate the profits for each product line for every month of each year. For this step, it would help to create new columns that indicate the profits, year, and month in the proper format to be used for the SUMIFS function.

Here are the subsequent SQL queries for each task:

SQL Query for Summary report 1:

```
SELECT      gp.productLine,
            SUM(ds.quantity) AS total_sales,
            TIMESTAMPDIFF(MONTH, MIN(ds.date), MAX(ds.date)) + 1 AS total_months,
            SUM(ds.quantity) / (TIMESTAMPDIFF(MONTH, MIN(ds.date), MAX(ds.date)) + 1)
            AS avg_monthly_sales
FROM        dailysales ds
JOIN        goproducts gp      ON      ds.productNumber = gp.productNumber
GROUP BY    gp.productLine
ORDER BY    gp.productLine;
```

This first task requires aggregating values taken from the very beginning of the dataset in terms of date, only splitting the results up by product line. Relevant data from different tables are merged with JOIN, and the ones to be eventually included in the report are taken by SELECT. In order to get the total number of months needed to calculate the average monthly sales, the TIMESTAMPDIFF function is used. Additional calculations for values that aren’t a column are specified in the SELECT clause.

SQL Query for Summary report 1A:

```
SELECT      gp.productLine,
            gp.productBrand,
            SUM(ds.quantity) AS total_sales,
            TIMESTAMPDIFF(MONTH, MIN(ds.date), MAX(ds.date)) + 1 AS total_months,
            SUM(ds.quantity) / (TIMESTAMPDIFF(MONTH, MIN(ds.date), MAX(ds.date)) + 1)
            AS avg_monthly_sales
FROM        dailysales ds
JOIN        goproducts gp ON ds.productNumber = gp.productNumber
GROUP BY    productLine, gp.productBrand
ORDER BY    productLine, gp.productBrand;
```

This SQL Query is essentially identical to the first report, the only difference being that it needs to be broken down further by product branch. To facilitate this, the data is grouped by both product line and product brand.

SQL Query for Summary report 1B:

```
SELECT      gp.productLine,
            YEAR(ds.date) AS year,
            MONTH(ds.date) AS month,
            SUM(ds.quantity * (gp.unitPrice - gp.unitCost)) AS total_monthly_profit
FROM        dailysales ds
JOIN        goproducts gp ON ds.productNumber = gp.productNumber
GROUP BY    gp.productLine, year, month
ORDER BY    gp.productLine, year, month;
```

The data is GROUP BY year and month in order to distinguish every individual month such that the total profit can be summed up.

Both the SQL and Excel approaches are both essentially doing the same calculations and processing, to return the same result for every generated report. The SUMIFS function in the excel process can be compared to the grouping and sum functions in the given SQL statements. Both methods return a summation of profit only from a filtered set of data. Both processes also had different methods for merge, but both just have the same goal of combining related tables of data for further processing.

The main difference of the 2 processes only came down to simplicity / cleanliness. The excel methods felt cluttered and less readable and in general, using SQL for anything related to data felt like a much more streamlined process.

In terms of OLAP operations, the Roll-up and Drill-down operations could be applied to generate the same results that were produced by the spreadsheet application and SQL queries. The Roll-up operation could be used for a summation of a quantity sold or total profit for example. Drill-down would be used for breaking it down to more specific levels such as by product line only, or down to every month.

Performance of the brands per country in terms of gross sales and net profit

To find the performance of the brands per country in terms of gross sales and net profit using Google Sheets, two additional sheets were created.

The first sheet facilitates the generation of additional attributes of each transaction in the `DailySales` sheets. In this sheet, data irrelevant to the task such as the `Order method code`, and `Date` are removed. After this, the `Retailer Country` and `Product Brand` are identified through the use of `VLOOKUP` functions having `Retailer Code` and `Product Number` of each transaction as its search key respectively. To get the `Gross Sales` resulting from each transaction, its `Quantity` and `Unit Price`, which are already present from the original data sheet, are multiplied. Lastly, to get `Net Profit`, which is the final column needed for the main report, the product of `Quantity` and the product's `Unit Cost`, which is obtained from the `Products` sheet using another `VLOOKUP` function, is subtracted from `Gross Sales`.

The second sheet consists of the results needed for the report. As a starting point for the results generation, the `UNIQUE` function is used to list the data grouped by `Retailer Country` and `Product Brand`. The rows generated by the aforementioned function are then used as the basis for the `SUMIF` function utilized to sum and present both `Gross Sales` and `Net Profit`. To be more specific, the `Retailer Country` and `Product Brand` are compared to all sales present in the first sheet. All the sales with matching `Retailer Country` and `Product Brand` are included in the summation of values to find the `Gross Sales` and `Net Profit` of each aggregated data.

To achieve the first subtask, a `Product` column is added to the first sheet which stores the `Product` retrieved by using another `VLOOKUP` function using the `Product Number` as the search key. As this column is separated from the `Retailer Country` and `Product Brand` column, a `FILTER` function, which determines which columns are considered, is nested in a `UNIQUE` function to list the aggregated data. All three columns are then used in a similar `SUMIF` function used in the first task to find the `Gross Sales` and `Net Profit` of each aggregate.

Different from the previous subtask, accomplishing the second subtask does not require any additional column to accomplish. Instead, this task requires another input specifically, the `Product Brand`, which will serve as the basis for determining whether a row is included in the result. To fulfill this task, a `FILTER` function, which compares the row's `Gross Sales` to the `Gross Sales` of the `Product Brand` input, determined by a `VLOOKUP` function, is used using the results of the main task as its basis.

To accomplish the main task using SQL statement, the multiplication operator `*` together with `JOIN` and `GROUP BY` clauses are used. The multiplication operator is used to calculate the `Gross Sales` of each aggregated data. On the other hand, the `JOIN` clause is used to determine both the `Retailer Country` and `Product Brand` of the data aggregated by the

GROUP BY function. After fulfilling this main task, performing the first subtask is done just by adding Product to the GROUP BY function. The second subtask however, entails the use of a subquery to determine the Gross Sales of the specified Product Brand which is then used in the WHERE clause of the main query to filter Product Brands with lower Gross Sales out. To be more specific, the following statements is used:

```
SELECT r.country, p.product_brand, SUM(s.quantity * s.unit_price) AS 'gross_sales', SUM(s.quantity * s.unit_price) - SUM(s.quantity * p.unit_cost) AS 'net_profit'
FROM daily_sales AS s
JOIN retailer AS r ON s.retailer_code = r.code
JOIN product AS p ON s.product_number = p.product_number
GROUP BY r.country, p.product_brand
```

SQL Query for Summary Report 2

```
SELECT r.country, p.product_brand, p.product, SUM(s.quantity * s.unit_price) AS 'gross_sales', SUM(s.quantity * s.unit_price) - SUM(s.quantity * p.unit_cost) AS 'net_profit'
FROM daily_sales AS s
JOIN retailer AS r ON s.retailer_code = r.code
JOIN product AS p ON s.product_number = p.product_number
GROUP BY r.country, p.product_brand, p.product
```

SQL Query for Summary Report 2A

```
SELECT r.country, p.product_brand, SUM(s.quantity * s.unit_price) AS 'gross_sales', SUM(s.quantity * s.unit_price) - SUM(s.quantity * p.unit_cost) AS 'net_profit'
FROM daily_sales AS s
JOIN retailer AS r ON s.retailer_code = r.code
JOIN product AS p ON s.product_number = p.product_number
GROUP BY r.country, p.product_brand
HAVING gross_sales > (
    SELECT SUM(s.quantity * s.unit_price)
    FROM daily_sales AS s
    JOIN product AS p ON s.product_number = p.product_number
    WHERE p.product_brand = 'X'
)
```

SQL Query for Summary Report 2B

Various similarities and differences are evident in doing these tasks using the two different methods. Using a spreadsheet application may generate extra columns, tables, or sheets to facilitate the processing of data which is absent in the SQL query method done to accomplish the same task; however, using SQL requires a more specialized knowledge in this programming language compared to using the more common spreadsheet applications. Additionally, utilizing a spreadsheet application may also provide users a different set of features such as conditional cell formatting which may be beneficial to users depending on their workflow. Overall, both methods led to the same results and using one over the other may just be a matter of preference.

The OLAP operations that can be applied to generate the report for subquery 2A is a *drill-down* operation while a *slice* operation may be used in fulfilling subquery 2B. The *drill-down* operation can be used in subquery 2A as from the original query where data is aggregated into product_brand, the query is modified to group data into product instead, which is lower in the concept hierarchy of the domain. On the other hand, the *slice* operation can be applied in subquery 2B as its result is the subset of the output of the original query in terms of one dimension only, namely in terms of gross_sales.


Performance of the retailers per country per quarter based on gross sales and net profit

To get the required report, the required tables are: DailySales, Products, and Retailers. The DailySales was used as the main table while the Products and

Retailers served as the Lookup Sheets. **Products** was merged with **DailySales** based on the column **ProductNumber**, introducing the **Cost** and **ProductName** columns to the main sheet. **Retailers** was merged with the main sheet using the **RetailerCode** attribute which appended the **RetailerName** and **Country** fields in the main sheet. A new column representing the **Quarter** field is appended to the table which parses the **OrderDate** column that is present on the original **DailySales** table. It simply gets the rounded up quotient of the month divided by 3 and the formula is implemented with the **ARRAYFORMULA** method which is built in google sheets.

The **GrossSales** is computed with **Quantity** sold multiplied by the **Price** column while the **NetProfit** is calculated by subtracting the **GrossSales** with the product of **Quantity** and **ProductCost**. Having accomplished this preprocessing of the new table, the Pivot Table function will be used to consider the constraints required by the specification. The relevant fields to be included in the rows attribute of the Pivot table are as follows: **Quarter**, **Country**, and **RetailerCode**. The fields to be included in the values attribute are: **GrossSales** and **NetProfit**. This approach generates the main query asked in the specification. The first subquery asks to show retailer's performance based on the quantity sold per product. This can be done in a similar fashion as the main query while only changing the attributes included in the Pivot table. The rows should have the **ProductNumber** included while the values can drop the **GrossSales** and **NetProfit**, replacing both with **Quantity**. The second subquery asks to show only the retailers who have a total quarterly gross sales of more than X amount. The pivot table used in the main query can be utilized for this query by just adding a filter and setting the condition to be greater than an input amount.

The SQL approach for the three queries have to merge the **DailySales**, **Products**, and **Retailers**. Similar with the method in excel, the **DailySales** is used as the main table and then the **JOIN** function is used to join the main table with **Products** on the **ProductNumber** column and with the **Retailers** on the **RetailerCode** column. The **QUARTER** function is used to obtain the quarter based on an input date. The **GrossSales** and **NetProfit** must be generated using the same formula used in the google sheets. To get the necessary summary of results, the query must be aggregated using the **GROUP BY** function according to **Year**, **Quarter**, **Country**, and **RetailerCode**. Then to clean the result format, the data can be sorted using the **ORDER BY** function. The second squery has similar structure as the first while dropping the **GrossSales** and **NetProfit** and adding the **ProductNumber** in the Group By function. The third query is nearly identical to the first query but a **HAVING** function must be utilized to set the condition for **GrossSales** to be greater than X amount. Attached below are the following SQL statements used.




```

SELECT YEAR(d.order_date) AS 'year', QUARTER(d.order_date) AS 'quarter',
r.country, r.retailer_name, SUM(d.quantity * d.unit_price) AS 'gross_sales',
SUM(d.quantity*d.unit_price) - SUM(d.quantity * p.unit_cost) AS 'net_profit'
FROM daily_sales d
    JOIN retailers r ON r.retailer_code=d.retailer_code
    JOIN products p ON p.product_number=d.product_number
GROUP BY YEAR(d.order_date), QUARTER(d.order_date),r.country,d.retailer_code
ORDER BY year,quarter,r.country,r.retailer_name ;

```

SQL Query for Summary Report 3




```

SELECT YEAR(O.order_date) AS 'year', QUARTER(O.order_date) AS 'quarter',
r.country, r.retailer_name, P.product_number, P.product_name, SUM(O.quantity) AS
total_quantity_sold
FROM daily_sales O
JOIN retailers R ON O.retailer_code = R.retailer_code
JOIN products P ON O.product_number = P.product_number
GROUP BY YEAR(O.order_date), QUARTER(O.order_date),r.country,O.retailer_code,
P.product_number
ORDER BY year, quarter, R.country, R.retailer_name, P.product_number,
total_quantity_sold;

```

SQL Query for Summary Report 3A



```

SELECT YEAR(d.order_date) AS 'year', QUARTER(d.order_date) AS 'quarter',
r.country, r.retailer_name, SUM(d.quantity * d.unit_price) AS 'gross_sales',
SUM(d.quantity*d.unit_price) - SUM(d.quantity * p.unit_cost) AS 'net_profit'
FROM daily_sales d
    JOIN retailers r ON r.retailer_code=d.retailer_code
    JOIN products p ON p.product_number=d.product_number
GROUP BY YEAR(d.order_date), QUARTER(d.order_date),r.country,d.retailer_code
HAVING gross_sales>1000000
ORDER BY year,quarter, r.country,r.retailer_name ;

```

SQL Query for Summary Report 3B

The use of excel provided a more graphical experience of manipulating the data as you can see the data as you select the functions that you need for processing. Both methods result to the same output when accomplished correctly. For this activity where the only task is to generate a report, the difference comes down to preference of the tool. However, when it comes to developing applications, SQL can be used more generally as it is a programming language that can be utilized using libraries in developing applications.

The OLAP methods notably used were the *drill down* and *dice*. The *drill down* method was used to *drill down* on the `OrderDate` column to obtain the `Quarter` column. The *dice* method was used to aggregate the data per Country by joining the `DailySales` table with the `Products` table based on the `ProductNumber` column.

Low performance

To generate the report for low performance in 2015 using Excel, several pre-processing steps and set operations were performed. First, it was necessary to denormalize the data to easily identify the `product` and `product brand` for each sale. This was done by merging `goDailySales` and `goProducts` with a right outer join on product number. After adding the table to the data model, we create pivot tables with `product` and `product brand` as rows. This creates columns of all distinct values from the original data. Assuming all `product` and `product brands` have made a sale, sets were made based on row items which will be used as the sets containing all `product` and `product brands`. Afterwards, the pivot tables were filtered with a timeline set to 2015 from which sets containing products and product brands with sales in 2015 were made. Finally, products and product brands with low performance in 2015 were generated by using the MDX function for set difference, `EXCEPT`, accordingly.

In SQL, the queries aim to retrieve distinct product names and product brands from the "products" table that were not sold in the year 2015, based on data from the "dailysales" table. The query uses a subquery to identify products that were sold in 2015 and then excludes those products from the final result set.

The subquery starts with a `RIGHT OUTER JOIN` between the "dailysales" table (aliased as "ds") and the "products" table (aliased as "p") on the common column "Product number." This join ensures that all products or product brands from the "products" table are included, even if they have no matching entries in the "dailysales" table.

The condition "`year(ds.date) = 2015`" filters the joined result set to only include entries from the year 2015. The `SELECT` clause in the subquery retrieves the product names or product brand associated with the sales entries in 2015.

The main query then selects distinct product names or product brands from the "products" table where the product or product brand is not found in the subquery result set. This ensures that only products not sold in 2015 are included in the final result. The `ORDER BY` clause arranges the results in ascending order based on the product names or product brands.

To be more specific, the following statements is used:

SQL Query for Summary Report 4A

```
SELECT distinct(product)
FROM products
WHERE product NOT IN
    (SELECT `product`
     FROM dailysales ds RIGHT OUTER JOIN products p
     ON p.`Product number`=ds.`Product number`)
```

```
WHERE year(ds.date) = 2015)
ORDER BY product ASC;
```

SQL Query for Summary Report 4B

```
SELECT distinct(`Product Brand`)
FROM products
WHERE `Product Brand` NOT IN
    (SELECT `Product Brand`
     FROM dailysales ds RIGHT OUTER JOIN products p
     ON p.`Product number`=ds.`Product number`
     WHERE year(ds.date) = 2015)
ORDER BY `Product Brand` ASC;
```

The SQL queries and Excel Functions generated the same set of low performing products and product brands in 2015, but they employ distinct methods, with each having its strengths and limitations.

The Excel approach may offer a more visual and interactive experience, especially for users comfortable with spreadsheet applications. It provides a step-by-step process, offering transparency in data manipulation. However, it could be resource-intensive for large datasets and may require additional expertise in Excel functions.

On the other hand, the SQL query offers a concise and efficient method for data retrieval, particularly when dealing with large datasets. It leverages the power of relational databases and SQL language, providing a straightforward solution for identifying products with low performance in 2015. However, it may be less visually intuitive for users not familiar with SQL.

In both SQL and spreadsheet applications, the OLAP operations of slice and dice can be applied to generate subqueries that focus on specific subsets of data, such as the data for the year 2015. In Excel, the slice operation can be achieved by using filters or pivot table slicers. After denormalizing and creating the pivot tables, you can apply a filter to the timeline or date column to include only the year 2015 data. In SQL, the slice operation can be implemented through a WHERE clause to filter data based on specific conditions. For example, the query retrieves data only for the year 2015, effectively slicing it to only include entries from the year 2015.

Discussion

In analyzing and generating summary reports from a large volume of data, two primary approaches were discussed by the group – the use of SQL queries and spreadsheet applications, specifically Excel. Each approach presents distinct advantages and considerations, contributing to a comprehensive understanding of the strengths and limitations associated with OLAP operations.

SQL queries are concise and clean, suitable for users with a database management background, offering efficient data manipulation. Excel, while potentially leading to a cluttered layout, is more accessible to a broader audience, providing transparency in data manipulation for users comfortable with spreadsheet applications.

SQL requires specialized programming knowledge, making it a barrier for some users, while Excel is widely used and accessible. SQL is efficient for handling large datasets, especially with complex queries and join operations. However, Excel may be resource-intensive for large datasets, leading to performance degradation during extensive calculations.

Excel offers a visual and interactive experience with charts, graphs, and pivot tables, suitable for users preferring a hands-on approach. SQL, while efficient, lacks the visual intuitiveness of Excel, typically presenting results in tabular form. Both methods support OLAP operations like roll-up, drill-down, slice, and dice, with Excel providing a user-friendly interface and SQL offering a more programmatic approach.

In conclusion, the choice between SQL and Excel depends on user needs, data nature, and personal preferences. SQL is efficient for handling large datasets and complex queries, while Excel is preferred for smaller datasets or by users comfortable with spreadsheet applications. Ultimately, both methods can achieve the same results and perform OLAP operations, and the decision depends on expertise and task-specific requirements.

Declaration

References

SHIPHYPE Fullfillment. (2022, January 5). *Average Monthly sales Definition*.

<https://shiphype.com/glossary-term/average-monthly-sales/>

Contributions

Name	Contribution
Corpuz, Joshua	Low Performance Section
Dimagiba, Rafael	<i>Average monthly sales, in terms of quantity sold, per product line</i> Section
Escopete, Steven Errol	<i>Performance of the brands per country in terms of gross sales and net profit</i> Section
Pineda, Ralph	<i>Performance of the retailers per country per quarter based on gross sales and net profit</i>