Universidade NOVA de Lisboa

Information Management School

# Adventure Works Bicycles SQL Project

*Managing Relational & Non-Relational Data*

*Author:*
Celso Cassamá
Student ID: 20222138

*Author:*
Guilherme Silva
Student ID: 20221209

*Author:*
Murilo Oliveira
Student ID: 20222137

*Author:*
Nuno Marques
Student ID: 20222145

2023

# Contents



# List of Figures



# List of Tables

# 1 Introduction

The project focused on the AdventureWorks2019 database, which contains information from the fictional bicycle manufacturer Adventure Works. The company primarily manufactures bicycles, however it also manufactures some of their own components. Other products sold by the company include components purchased from other manufacturers, accessories and clothing.

The company has a global presence, being present in countries such as the United Sates, Australia and the United Kingdom, among others. Their customer base is divided into two groups, retail stores and individual customers. Despite not possessing any physical locations, Adventure Works sells their own products via an Internet presence. The company also sells in bulk to retail distributors, which in turn resell their products.

The aim of this project was to solve a series of business problems using procedures to make changes to the database in order to address the limitations and implement new features, as described in the project specifications.

The project addresses two specific business problems related to Adventure Works' stock related issues and the possibility of expansion into first-party retail locations. These will be further delved into in their respective sections, however they both require the study of the company database in order to create additional functionality and help in the decision of the location of the possible retail locations, respectively.

# 2 Stock Clearance

This problem stemmed from the recurring issue of considerable old bicycle stock at Adventure Works Bicycles, which affects the successful launch of new models. An aggressive discount campaign in the previous year failed to resolve the issue.

Therefore, the company planned to implement an online auction for all products for which a new model is expected to be announced in the near future.

The task at hand required the extension of the current database in order to support the auctions. This included both the required tables and columns and the procedures to operate on said elements.

## 2.1 Auction Schema

The first step towards creating the required infrastructure was the creation of the *Auction* schema. The process started by ensuring that no existing schemas named auction existed. After verifying that *Auction* did not already exist, a new schema with said name was created with the following tables: *AuctionedProducts*, *Bids* and *Threshold*.

*Auction.AuctionedProducts* stored information regarding the products being auctioned. This included their auction ID, which was used as the primary key, product ID, starting and end dates for the auction, initial bid, status and winning bid.

*Auction.Bids* contained information about all valid bids placed on the products, such as the bid ID, which was used as the primary key, the ID of the customer placing the bid, the ID of the auction, the amount bid and the time at which the bid was placed.

*Auction.Threshold* recorded the information pertaining to the global configuration for the bids and auctions. This consisted of data such as the threshold ID, which was used as the primary key, the minimum value increase for the bids, the maximum auction price, the minimum starting date and maximum expiration date for the auction.

The updated database schema can be seen in figure 1.

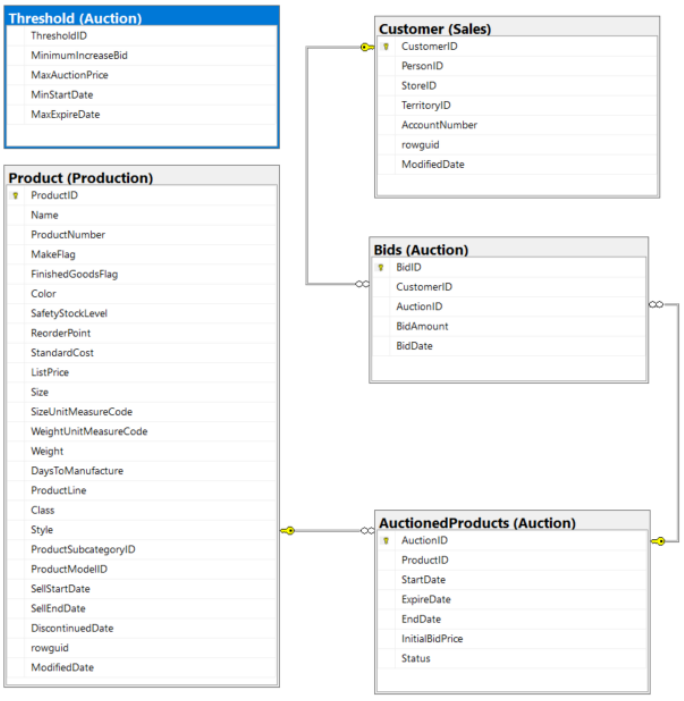


Figure 1: Updated database schema

With the necessary tables created, the next step was developing the procedures that would be used

to extract and alter information in said tables.

### 2.2 Auction.uspAddProductToAuction

This procedure aims to enable the addition of new products. It takes in as parameters the product ID, the auction expiration date and the initial bid. Firstly it checks if the product being added exists, is on sale and has stock to be cleared. If the product meets all the required conditions it then checks if the product has already been added to the *AuctionedProducts* table.

If the product has not already been added, the procedure then fetches the necessary information from *Threshold*, including start and end date limits, minimum bid increase and maximum auction price for the designated product.

Afterwards, a series of checks regarding starting and expiration checks are performed and, if necessary, adjustments are made or, if not possible, errors are raised. These including checking if the expiration date is prior to the current date.

A similar procedure for the initial bid price is done, checking if the provided input is valid and correcting or informing the user of the error. The flag parameter, that differentiates between products manufactured in-house or those that are simply resold by the company, is then used to set the minimum bid price.

After performing all the necessary verifications and informing the user of possible errors, if all the information provided is valid the procedure then adds the product to *AuctionedProducts*.

### 2.3 Auction.upsTryBidProduct

The procedure ensures that potential bids pass the required checks before adding them to the *Bids* table. It takes in as parameters product ID, customer ID and the bid amount.

Firstly, a check is made to ensure that the customer ID associated with the bid exists, exiting the procedure if not. It also verifies that the product that is being bid on is being auctioned, exiting if not. Finally, a check regarding the date is performed, exiting if the bid is too early for the product in question.

If all of the first three checks are passed, the procedure then checks if a bid amount is specified by the user and is larger than the maximum current bid, adding the minimum increase to the maximum existing bid if not and setting that value as the bid amount.

Another check is then performed regarding the bid amount and the maximum price for the product in question, which is determined using the product retail value and the threshold present in *Threshold*, checking if the former is smaller than the latter.

If all checks are passed the bid is then inserted into *Bids* with the customer ID, auction ID, bid amount and current time.

### 2.4 Auction.upsRemoveProductFromAuction

The procedure removes a product from auction. It takes in as parameters the product ID.

Firstly, it checks if the product is currently being auctioned by querying *AuctionedProducts*, exiting if the product is not found.

If the product is found, the procedure updates the status and EndDate columns, setting the first to 'Canceled' and the second to the current date and time.

The procedure does not return any sets, but it may raise errors if the input is invalid or a problem arises when updating the database.

### 2.5 Auction.upsListBidOffersHistory

The stored procedure takes four input parameters, including the customer ID, auction starting and end time and the flag to see if a product has an active auction. It checks if the specified customer ID has ever placed a bid on any product in the *Bids* table, and if there are no records, it raises an error message and returns control to the calling program without executing any further code.

If the Active parameter is set to 1, the procedure retrieves the bid offers history for the specified customer ID within the specified date range, where the corresponding auctioned product is still active. On the other hand, if the Active parameter is not set to 1, the procedure retrieves the bid offers history for the specified customer ID within the specified date range, regardless of the status of the corresponding auctioned product.

The procedure returns the retrieved bid offers history. It joins the *Bids* and *AuctionedProducts* tables and selects the relevant data based on the input parameters. This stored procedure can be used to track the bid history of a customer within a specified time frame and determine if they were successful in winning any auctioned products.

### 2.6 Auction.uspUpdateAuctionStatus

The stored procedure updates the status of an active auction to "Closed" and sets the end date of the auction to the current date and time. It also sets the WinnerBid to the bid ID of the customer who placed the highest bid in the auction. This ensures the proper closing of auctions by updating the relevant fields and ensures data consistency through the use of transactions and error handling.

To ensure data integrity, the procedure first starts a transaction using the BEGIN TRANSACTION statement, and then executes an UPDATE statement to change the status of the active auctioned products that have expired based on the ExpireDate field. The subquery in the UPDATE statement finds the highest bid placed by a customer in the auction and sets the WinnerBid field of the corresponding auction product to the bid ID of the winning bid.

To handle errors and ensure data consistency, the procedure checks the transaction by rolling back the transaction using the ROLLBACK TRANSACTION statement if an error occurs during the transaction. The procedure also prints the error message and throws a custom error message using the THROW statement.

## 3 Brick and Mortar Stores

This problem consisted in assisting Adventure Works regarding the decision of the location of their first two brick and mortar stores. These would provide a direct selling avenue for the company, avoiding the usage of intermediaries and their associated fees. The location of these stores had some inherent restrictions, such as being located in the United States and avoid cities where they would directly compete with their existent re-sellers. As such, the cities containing their best 30 customers were excluded.

Opening physical points of sale, requires the deployment of capital, to cover starting costs, that is expected to be paid back with interest in a certain time frame. One of the most important financial metric when evaluating an investment project is Internal Rate of Return (IRR), which measures the rate of return of an investment discounting the time value of money. The interest generated by the investment is more valuable the sooner it is generate, therefore, when recommending the top two cities where to open the brick and mortar stores, an option was made to favor those with the highest Gross Margin. Assuming marketing, sales, logistics and overheads costs are shared equally across the organization, the cities where costumers buy products with higher Gross Margins are a good proxy to identify where the company is more profitable and ultimately has a better IRR.

The first step in making an informed suggestion was defining the criteria by which possible locations would be evaluated. These factors were chosen based both on the requirements established in the problem description, as well as those that would give the best indication of possible future profits, as the goal of any venture is to maximize profit. As such the final criteria, only taking into account the past 12 months, were:

- High Total Gross Margin (Unit Price-Standard Cost).
- Being located in the US.
- Not being located in the same cities as the top 30 re-sellers.

### 3.1 Queries for obtaining relevant information

With the criteria for evaluation defined, the next step was extracting relevant information from the database.

The query, as seen in annex 1, selects the top two cities where individual customers are located based on the sum of the Total Gross Margin of their orders. It joins several tables, including SalesOrderHeader, SalesOrderDetail, Customer, Person, BusinessEntityAddress, Address, StateProvince, and Product to extract the required data.

It then filters only orders placed by customers located in US cities and active products sold in the last 12 months, excluding canceled and rejected orders. It also excludes the top 30 cities where the store contacts are located. The cities are grouped by their names, and the sum of the difference between UnitPrice and StandardCost for all orders is calculated for each city. Finally, the cities are ordered in descending order of their Total Gross Margin, and the top two cities are returned as the result of the query.

The output of the query is table 1, which contains the top two cities by Total Gross Margin, excluding those that fail to meet the other criteria.

Table 1: Gross Margin by City

| **City** | **TotalGrossMargin** |
|---|---|
| Bellflower | 69153,1182 |
| Berkeley | 58867,4499 |

As seen in table 1, the prime locations for the creation of brick & mortar stores would be Bellflower and Berkeley, according to the previously established criteria.

# 4 Annexes

Listing 1: Brick and Mortar Query

```
SELECT TOP 2 a.City, SUM(sod.UnitPrice - prod.StandardCost) AS TotalGrossMargin
from Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod ON sod.SalesOrderID = soh.SalesOrderID
JOIN Sales.Customer c ON c.CustomerID = soh.CustomerID
JOIN Person.Person p ON p.BusinessEntityID = c.PersonID
JOIN Person.BusinessEntityAddress bae ON bae.BusinessEntityID = p.BusinessEntityID
JOIN Person.Address a ON a.AddressID = bae.AddressID
JOIN Person.StateProvince sp ON sp.StateProvinceID = a.StateProvinceID
JOIN Production.Product prod ON prod.ProductID = sod.ProductID
WHERE sp.CountryRegionCode = 'US' AND OrderDate >=
DATEADD(year, -1, (SELECT MAX(OrderDate) FROM Sales.SalesOrderHeader))
AND prod.SellEndDate IS NULL AND soh.Status in (1,2,5)
AND a.City NOT IN
(
SELECT TOP 30 a.City
from Sales.SalesOrderHeader soh
JOIN Sales.Customer c ON c.CustomerID = soh.CustomerID
JOIN Sales.Store s ON s.BusinessEntityID = c.StoreID
JOIN Person.BusinessEntityAddress bae ON bae.BusinessEntityID = s.BusinessEntityID
JOIN Person.Address a ON a.AddressID = bae.AddressID
JOIN Person.StateProvince sp ON sp.StateProvinceID = a.StateProvinceID
WHERE sp.CountryRegionCode = 'US' AND soh.Status in (2,5)
GROUP BY s.BusinessEntityID, a.City
ORDER BY SUM(soh.SubTotal) DESC
)
GROUP BY a.City
ORDER BY TotalGrossMargin DESC
```