Project 3: Prestige Cars Normalized Database

Members:

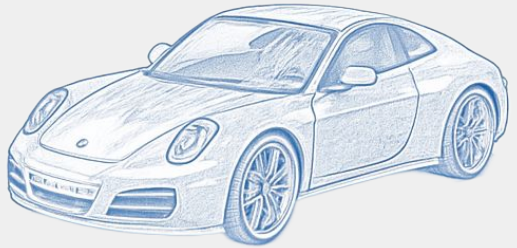


TABLE OF CONTENTS

01: Project Overview

Main Objective: 

- Transform the Prestige Cars dataset into a relational database that is organized and logically structured.
- Apply industry-standard modeling methods, as well as proper normalization practices.
- Demonstrate how business rules, data governance, and data-quality controls are enforced throughout our database.

✓ The How’s:

- Clean the given dataset and normalize it into proper tables and schemas.
- Establish primary / foreign key relationships to ensure data integrity.
- Explain why we designed our SQL Server database the way we did.
- Deliver a database that is practical, consistent, and well-structured

02: Original Dataset Issues

Original

SchemaName	TableName	ColumnName	OrdinalPosition	DataType	IsNullable
Data	Customer	IsReseller	8	bit	YES
Data	Customer	IsCreditRisk	9	bit	YES
Data	Stock	IsRHD	7	bit	YES
Data	Make	MakeCountry	3	char(3)	YES
Data	Model	YearFirstProduced	5	char(4)	YES
Data	Model	YearLastProduced	6	char(4)	YES
Data	Sales	InvoiceNumber	3	char(8)	YES
Data	Stock	DateBought	10	date	YES
Data	Sales	SaleDate	5	datetime	YES
Data	Sales	ID	6	int	NO
Data	Sales	SalesID	1	int	NO
Data	SalesDetails	SalesDetailsID	1	int	NO
Data	SalesDetails	SalesID	2	int	YES
Data	Stock	Cost	3	money	YES
Data	Stock	RepairsCost	4	money	YES
Data	Stock	PartsCost	5	money	YES
Data	Stock	TransportInCost	6	money	YES
Data	Customer	Country	7	nchar(10)	YES
Data	Country	CountryISO2	2	nchar(10)	YES
Data	Country	CountryISO3	3	nchar(10)	YES
Data	Country	FlagFileType	7	nchar(3)	YES
Data	Sales	TotalSalePrice	4	numeric(10, 2)	YES
Data	PivotTable	2015	2	numeric(10, 2)	YES
Data	PivotTable	2016	3	numeric(10, 2)	YES
Data	PivotTable	2017	4	numeric(10, 2)	YES
Data	PivotTable	2018	5	numeric(10, 2)	YES
Data	SalesDetails	SalePrice	5	numeric(10, 2)	YES
Data	SalesDetails	LineItemDiscount	6	numeric(10, 2)	YES
Data	Make	MakeName	2	nvarchar(100)	YES
Data	Customer	CustomerName	2	nvarchar(150)	YES
Data	Model	ModelName	3	nvarchar(150)	YES
Data	Model	ModelVariant	4	nvarchar(150)	YES
Data	Country	CountryName	1	nvarchar(150)	YES
Data	Country	SalesRegion	4	nvarchar(20)	YES
Data	Stock	BuyerComments	9	nvarchar(4000)	YES
Data	Customer	CustomerID	1	nvarchar(5)	NO
Data	Sales	CustomerID	2	nvarchar(5)	YES
Data	SalesDetails	StockID	4	nvarchar(50)	YES
Data	Stock	Color	8	nvarchar(50)	YES
Data	Stock	StockCode	1	nvarchar(50)	YES
Data	Customer	Address1	3	nvarchar(50)	YES
Data	Customer	Address2	4	nvarchar(50)	YES
Data	Customer	Town	5	nvarchar(50)	YES
Data	Customer	PostCode	6	nvarchar(50)	YES
Data	Country	FlagFileName	6	nvarchar(50)	YES
Data	PivotTable	Color	1	nvarchar(50)	YES
Data	Make	MakeID	1	smallint	NO
Data	Model	ModelID	1	smallint	NO
Data	Model	MakeID	2	smallint	YES
Data	Stock	ModelID	2	smallint	YES
Data	Stock	TimeBought	11	time	YES
Data	SalesDetails	LineItemNumber	3	tinyint	YES
Data	Country	CountryFlag	5	varbinary	YES

Issue 1: Schema Names

The original dataset lacked categorization, all tables were placed under one generic schema, which makes it messy and unorganized, and ultimately failing to implement data governance from a business standpoint.

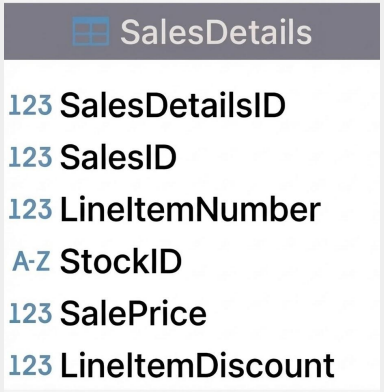
SchemaName	TableName	C
Data	Customer	Is
Data	Customer	Is
Data	Stock	Is
Data	Make	M
Data	Model	Y
Data	Model	Y
Data	Sales	Ir
Data	Stock	D
Data	Sales	S
Data	Sales	ID
Data	Sales	S

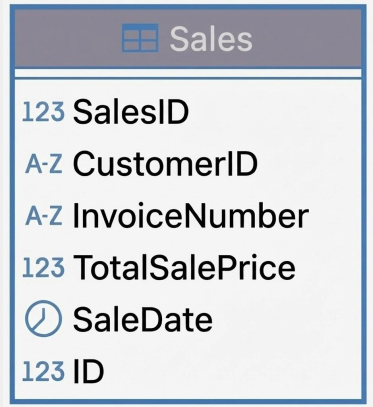
Data	Customer	
Data	Country	
Data	Country	
Data	Country	
Data	Sales	
Data	PivotTable	
Data	PivotTable	
Data	PivotTable	
Data	PivotTable	
Data	SalesDetails	
Data	SalesDetails	
Data	Make	
Data	Customer	

Issue 2: Misleading Table Names

There were some tables that didn't really represent the data they contained, which can cause confusion.

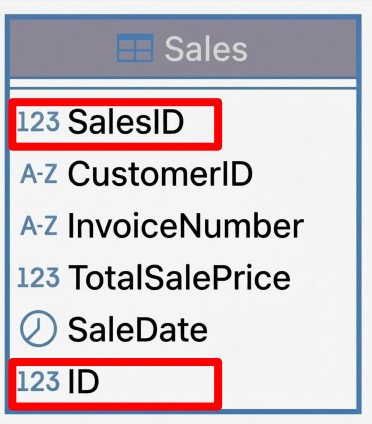
For example, the Sales and SalesDetails tables were not storing sales revenue or yearly sales details but instead had customer order details.



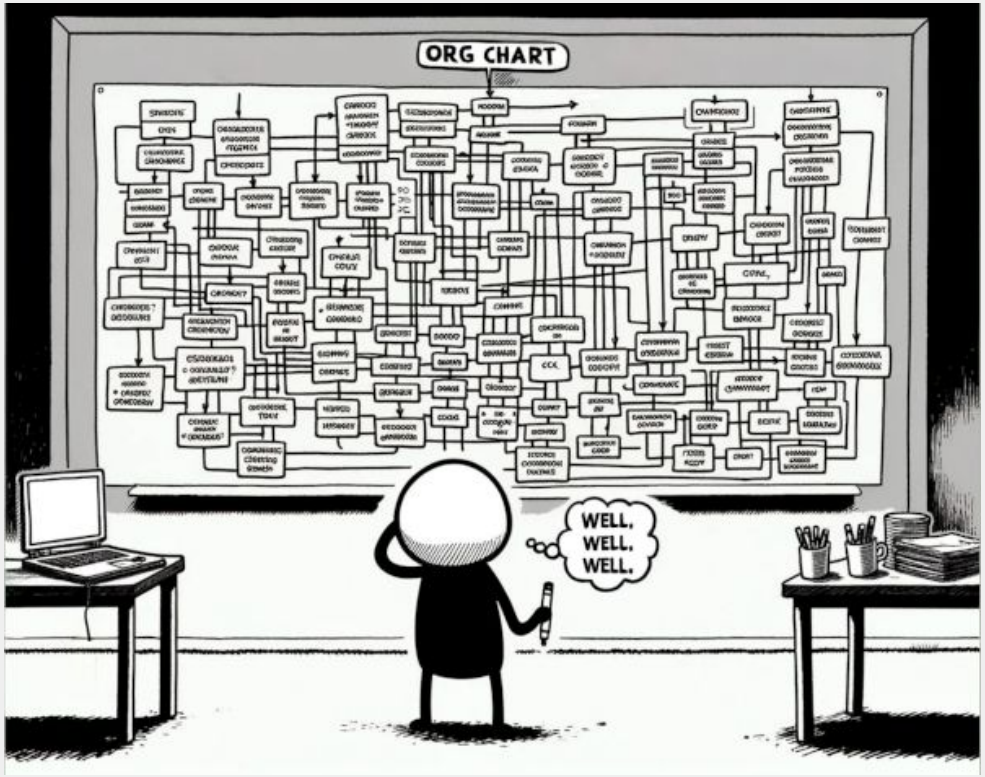


Issue 3: Unclear Relationships

The original dataset had unclear table relationships mainly because the primary and foreign keys weren't properly defined. Without these keys, the tables can't be linked correctly or correlated with each other. One table even had two ID columns which shows how inconsistent the structure was.



03: Database Implementations & Adjustments



Schema Changes

Schemas created:

- Sales
- Cars
- Data

Each schemas represent a specific focus area, that allows tables to be grouped by purpose instead of being placed under one generic schema. This separation makes it clear what each table relates to and helps identify and navigate information more quickly. Organizing the database this way improves clarity, reduces confusion, and keeps the overall structure easier to understand and maintain.

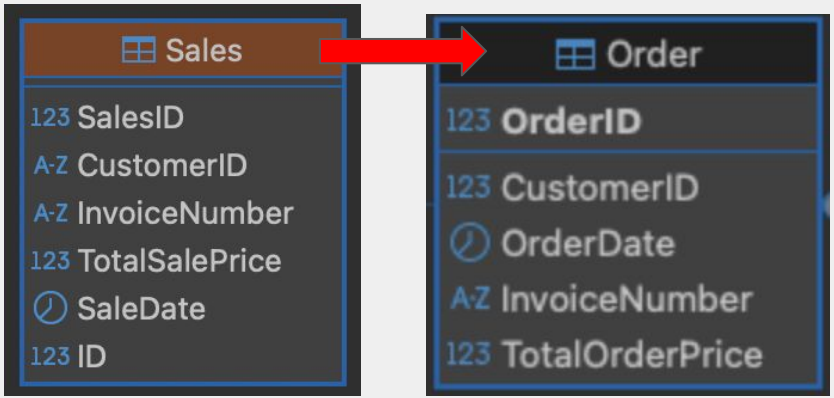
Tables Created

We created the Sales.Address table because the original Customer table was storing all the address fields inside it. This violates the normalization rules because it was mixing two different types of information together, making the data harder to manage.

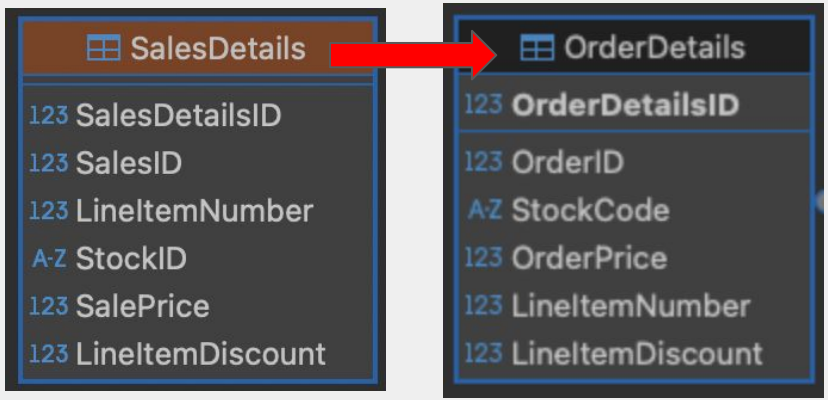
By moving the address fields into their own table and linking it back to Customer with a foreign key, the data becomes more organized, avoids repeated addresses, and makes it easier to update and manage.

Tables Renamed

Sales → Orders



SalesDetails → OrderDetails



Primary Keys

In our database, we added identity-based primary keys that automatically generates a unique number for every new row.

We applied a consistent naming pattern such as PK_Country_CountryID and PK_Order_OrderID, so that it's easier to see which table each key belongs to.

These identity-based primary keys ensure that every row is unique, allows us to build correct relationships between tables using foreign keys, and help create a clean, stable foundation for the entire database.
(unique, non-nullable, identifier)

Identity-Based Primary Keys

In our database, we used Identity columns to automatically create unique ID numbers for each new row.

Meaning that the database generates the IDs on its own (1, 2, 3, and so on), so we don’t have to enter them manually.

This helps prevent mistakes like duplicate or missing IDs, or manually entering incorrect values.

We applied IDENTITY(1,1) to primary key fields because the original dataset did not have a reliable way to generate unique IDs on its own.

And so adding this just keeps the data consistent and makes inserting safer and easier.

Foreign Keys

In our database, we added foreign key constraints across all dependent tables, such as linking Orders → Customers, OrderDetails → Orders, Address → Country and Customer, etc, to help support accurate joins and improve data reliability.

These constraints were necessary because the original dataset referenced these relationships but never formally enforced them.

To clearly document and identify each relationship we used the Format: FK_ChildTable_ParentTable_ColumnName

Alternate keys

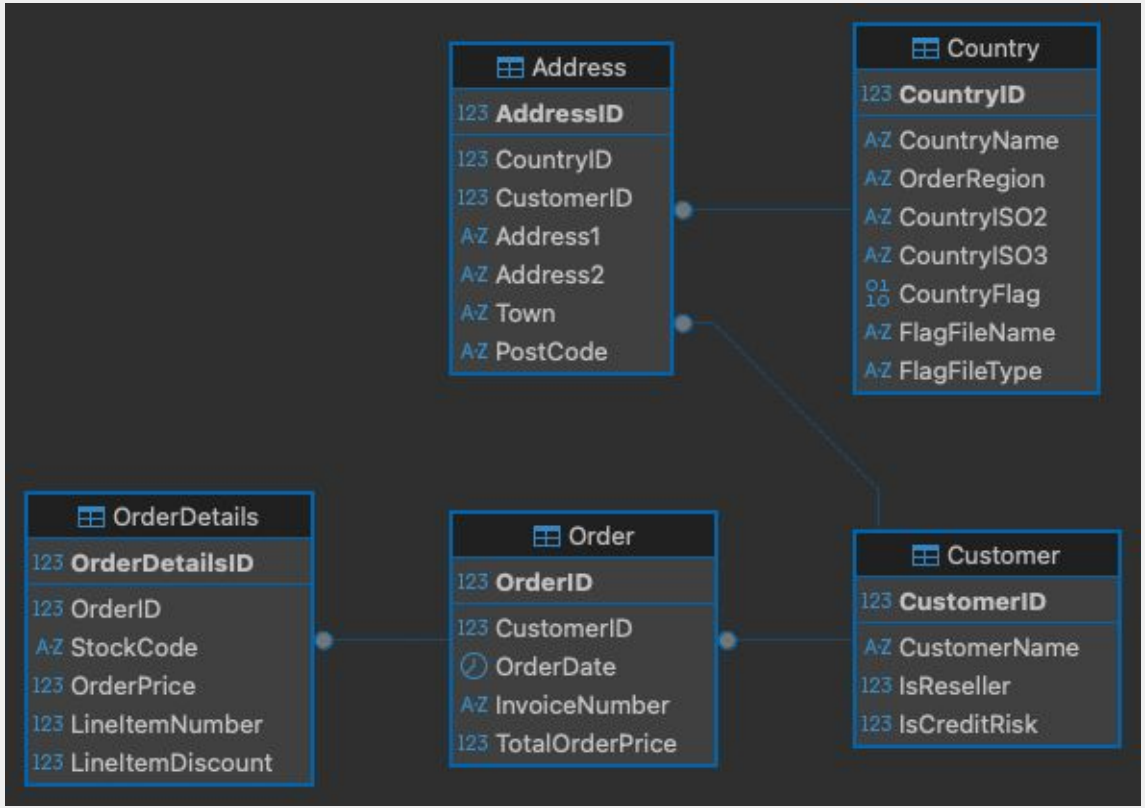
In our database, we added Alternate Keys to make sure certain columns stay unique, even if they are not primary keys.

We used the naming pattern AK_Table_Column to make it clear which column is being kept unique.

For example, columns like StockCode and InvoiceNumber must always be unique because they identify one specific item or one specific invoice.

These constraints help the system automatically block any duplicate values when new data is added, keeping the information clean and reliable. Overall, alternate keys improve the accuracy of the data and prevents confusion or errors caused by duplicates.

Mapping of Keys (Diagram)



Check Constraints

In our database, we added Check Constraints to make sure that values going into certain columns are valid and follow real-world rules.

We used the naming pattern CK_Table_Column so that it’s easy to see exactly which column the rule applies to.

For example, things like costs, repair costs, or the year a model was first produced should never be negative or unrealistic.

In the original dataset, there was no check constraint that stopped someone from entering a negative cost or year, which could lead to inaccurate calculations and incorrect reporting.

Overall, check constraints help catch mistakes early, block invalid data from being saved, and keep the entire database more accurate. They protect data quality before small issues turn into bigger problems.

Default Constraints

In our database, we added Default Constraints to make sure certain columns automatically receive a value when a new row is inserted.

This prevents important fields from being left blank and keeps the data consistent.

For example, OrderDate now automatically fills in with the current date and time whenever a new order is created, so that we don’t have to manually insert one.

We also gave IsReseller and IsRHD default values so they always start with a clear indicator (such as 0 or 1) instead of being left empty. These automatic defaults reduce user mistakes, prevent NULL values, and makes sure new data begins with reliable, meaningful information.

Format: DF_Table_Column

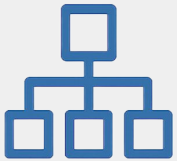
Indexes

In our database, we added Indexes to make certain searches and lookups much faster.

In the original dataset, columns like CountryName and Address fields didn't have any indexes to speed up those searches. As data grows, the database can become slow and inefficient.

By creating indexes such as IX_Country_CountryName_CountryISO2_CountryISO3 and IX_Address_AddressLine1_AddressLine2_Town_PostCode, we make these lookups significantly faster and reduce the load on the database.

Format: IX_TableName_ColumnName (or IX_TableName_Column1_Column2 when multiple columns are indexed)



04: UDTs & FQDNs

What is a UDT?

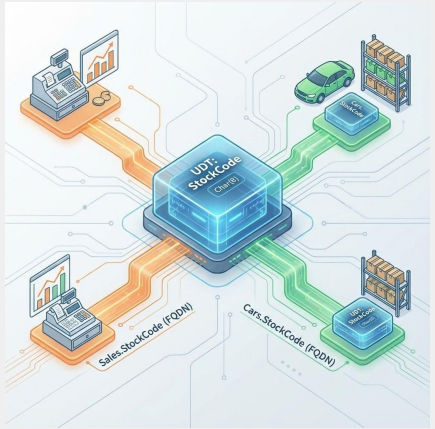
A User-Defined Data Type is a custom data type that we create using an existing SQL data type, but with our own set of constraints or rules.

Why use UDTs?

- Promotes code reusability: can be used across multiple tables and databases
- Useful when dealing with complex data
- Helps maintain data integrity

https://www.sqlshack.com/an-overview-of-user-defined-sql-server-types/

https://www.selectdistinct.co.uk/2024/03/06/explaining-user-defined-types-in-sql-server/



Sales.StockCode (UDT)

We created a UDT for StockCode because this value appears in more than one table and represents the same business concept across the database.

When we define the datatype once in the UDT, every table that uses StockCode will automatically follow the same structure, rules, and meaning. This helps keep the data consistent and prevents errors when joining or comparing tables.

The UDT's TypeName also reinforces the business meaning because it's named after the real idea of a "Stock Code," and it's clear to anyone reading the database what kind of value belongs there.

Cars.StockCode (UDT)

We applied the same UDT to Cars.StockCode since this value identifies each vehicle in the inventory, and it needs to be stored in the same way as the Sales schema.

Using the UDT here makes sure that every vehicle's Stock Code follows the same datatype, length, and rules as the Stock Codes stored in the Sales schema.

This establishes a connection between Cars and Sales data and avoids having mismatched formats or inconsistent entries.

Furthermore, the UDT's TypeName makes the business purpose obvious, since any column using this type is expected to store a real Stock Code, not just any random text.

Overall, applying said UDT to Cars.StockCode ensures that every vehicle identifier is stored the same way and displays the same business concept throughout the database.

What are FQDNs?

A Fully Qualified Domain Name means that the schema name is actually a business defined category. This domain name represents a group of information that makes sense to a business or company, like Sales Information, Cars Information, Customer Information, etc.

So when we use a domain like Sales.StockCode or Cars.StockCode, the name tells us what kind of information it belongs to and what it represents.

In our database, the two main domains we used were Sales and Cars, both using the same StockCode UDT that follows the same rules, structure, and business meaning. Using FQDNs helps keep the related data organized by business purpose and guarantees that any column using the FQDN follows the same semantics throughout the database..

05: Database Interfaces

Views and ITVF’s

A view is a stored query in the database. When queried, it executes that saved query and returns its results. It acts as a virtual table, therefore it does not actually store any data instead their definitions are stored as permanent objects in the database..

An inline table valued function is an example of a udf, user defined function, this means it returns a table data type. It is similar to a View except that it is parameterized, making it more flexible to filter or change results.

Views and ITVF’s both support the logical layer of the database, providing users with a clean and logical way to interact with database.

Views/ITVF’s Created

DataTransfer.v_AllSales

CREATE OR ALTER VIEW DataTransfer.v_AllSales AS

SELECT 2015 AS SaleYear, * FROM Sales2015

UNION ALL

SELECT 2016 AS SaleYear, * FROM Sales2016

UNION ALL

SELECT 2017 AS SaleYear, * FROM Sales2017

UNION ALL

SELECT 2018 AS SaleYear, * FROM Sales2018;

This view combines all yearly sales tables into one virtual table. Instead of querying 4 separate tables, it gives you one dataset.

Views/ITVF’s Created

DataTransfer.v_SalesByMake

```
CREATE OR ALTER VIEW DataTransfer.v_SalesByMake AS
SELECT
    SaleYear,
    MakeName,
COUNT(*) AS CarsSold,
    SUM(SalePrice) AS Revenue,
    SUM(SalePrice -
        (ISNULL(Cost,0) + ISNULL(RepairsCost,0) + ISNULL(PartsCost,0) + ISNULL(TransportInCost,0))
    ) AS Profit
FROM DataTransfer.v_AllSales
GROUP BY SaleYear, MakeName;
```

This view summarizes sales by SaleYear and MakeName, making it easier to run reports that can be grouped by car brand.

Views/ITVF’s Created

Reference.v_BudgetSummary

```
CREATE OR ALTER VIEW Reference.v_BudgetSummary AS
SELECT
    Year,
    Month,
    BudgetElement,
    SUM(BudgetValue) AS TotalBudgetValue,
    COUNT(*) AS BudgetItems
FROM Reference.Budget
GROUP BY Year, Month, BudgetElement;
```

This view summarizes budget numbers by Year, Month, and BudgetElement.

Views/ITVF’s Created

Reference.f_GetBudgetByYear

```
CREATE FUNCTION Reference.f_GetBudgetByYear
(@Year INT)
RETURNS TABLE
AS RETURN
(   SELECT *
    FROM Reference.Budget
    WHERE Year = @Year
);
```

This ITVF returns all budget rows for a specific year, again making it easier for reporting.

Views/ITVF’s Created

Reference.f_SalesByYear

```
CREATE FUNCTION Reference.f_SalesByYear
(@Year INT)
RETURNS TABLE
AS RETURN
(  SELECT *
    FROM Reference.YearlySales
    WHERE YEAR(SaleDate) = @Year
);
```

This ITVF returns sales data for a chosen year from the Yearly Sales table, which can help analyze performances.

06: Conclusion

Conclusion

- Reorganized the original dataset into clear business schemas (Sales, Cars, Data)
- Added identity PKs, FK relationships, alternate keys, check constraints, defaults, and indexes for better integrity and performance
- Applied UDTs and FQDNs to enforce consistent business meaning across tables
- Final design is normalized, structured, reliable, and supports accurate reporting and analytic