Relational databases group process log

DATA PROCESSING
SEMESTER 4 2020/2021

Lex Van Dalen,
Slawomir Twardowski
Docent Jeroen De Kort

INTRODUCTION

In this document we archive the iteration process of our group assignment project during Semester 4 concerning relational databases.

This project includes designing a database on the basis of delivered data files and requirements described by the scenario; developing software required to clean-up and copy given data into the newly designed database; and finally visualizing the data in a dashboard, made with software package chosen by the group.

On the following pages you'll find screenshots and descriptions illustrating the process and choices made with explanation of motivations behind our choices.

CONTENTS

DESIGN

After supplied files and Domino webpage analysis (which is documented in a separate document concerning analysis of the project) we begin by drawing first versions of DB diagrams. For this purpose, we decided to use DRAW.IO. Our choice here was motivated by the simplicity and availability of the tool and it's version control possibilities (easy and free access to GitHub). In this case the simpler the better since being able to quickly iterate and share the changes between group members was key.

1. Illustrating the data within the supplied Mario Data files

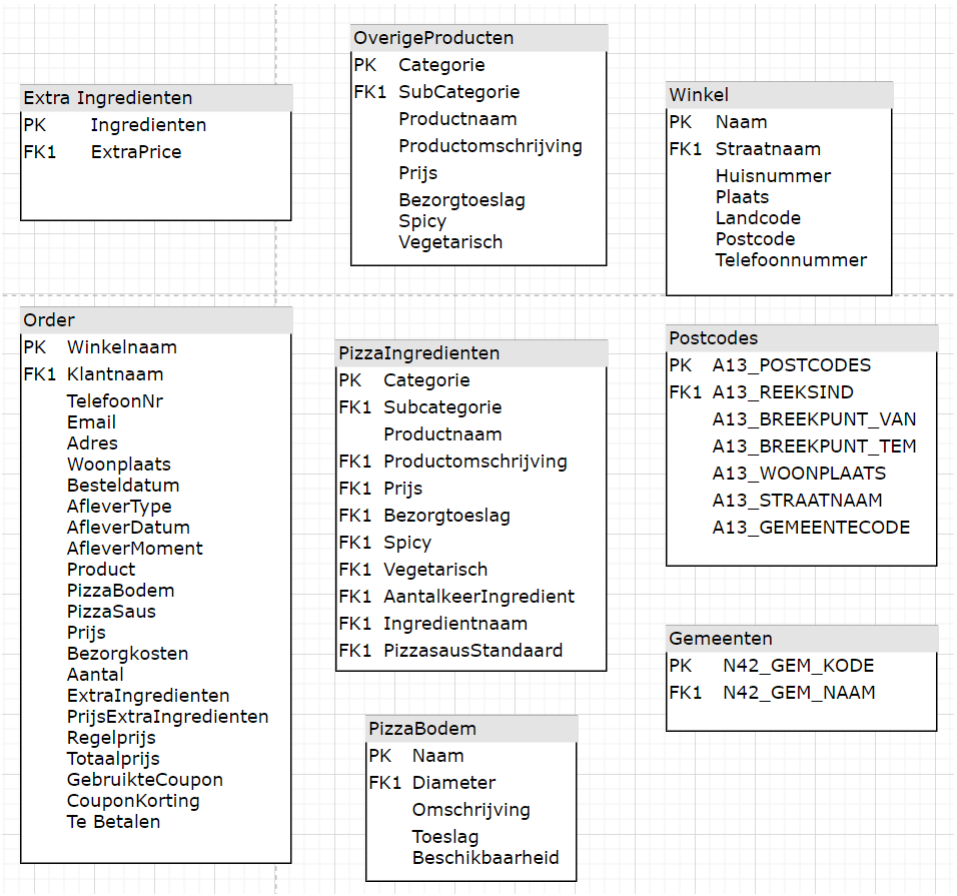

Fig. 1- Diagram mirroring existing files

Our first commit is just a simple illustration of the tables supplied within the Mario Data files. From this point on we can think of arranging the data in a way that would best suit the requirements that came forth from the analysis (PK/FK are not properly applied and should be ignored at this stage, since the data will be completely re-arranged anyways)

2. Re-arranging the data and deciding on data-types

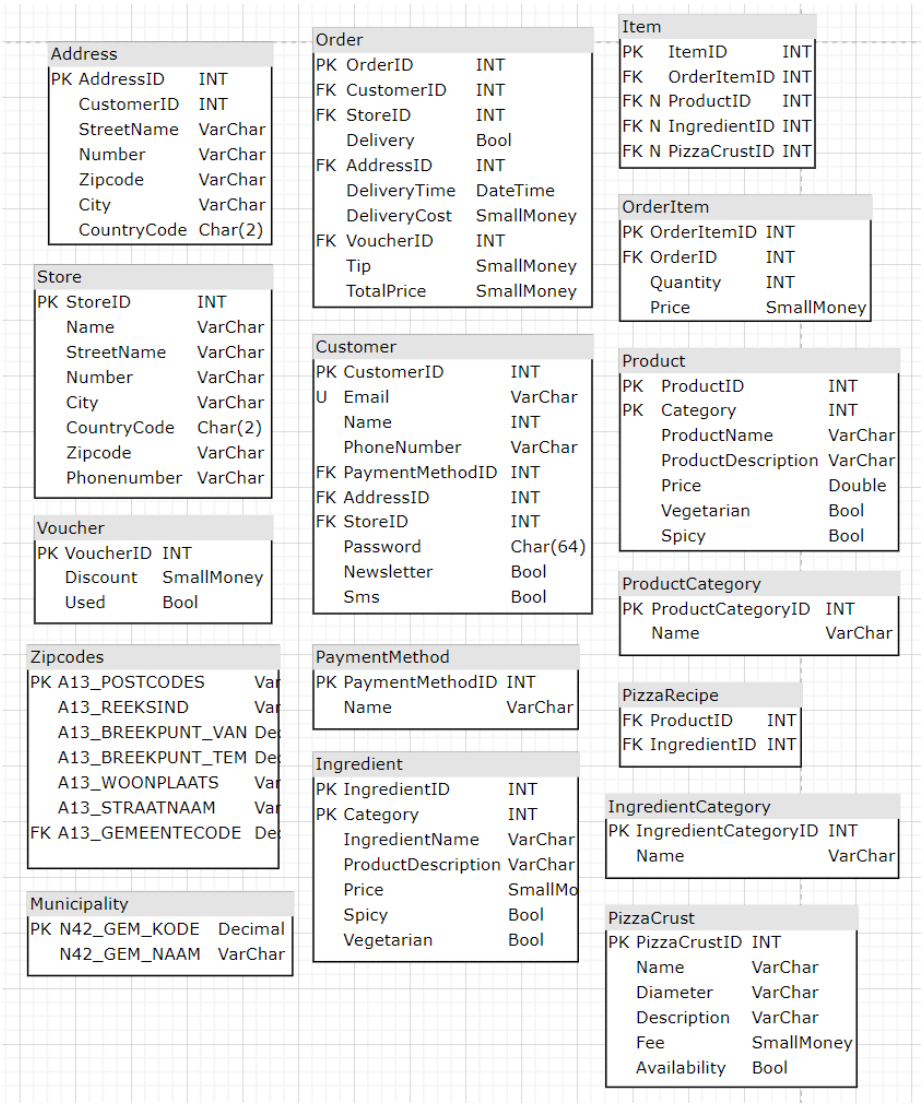

Fig. 2 - Data re-arrangement

The following step in the project is re-arranging the data into new tables of our own design and deciding upon the data types used in the DB. We have also set Primary and Foreign Keys.
We intend to use SmallMoney type (SQL Server) where money amounts are concerned, INT type for PK/FK, Date or Time for fields concerning time and VARCHAR of varying length to accommodate names, descriptions and any fields where numbers are mixed with alphabet.

3. Tables connections

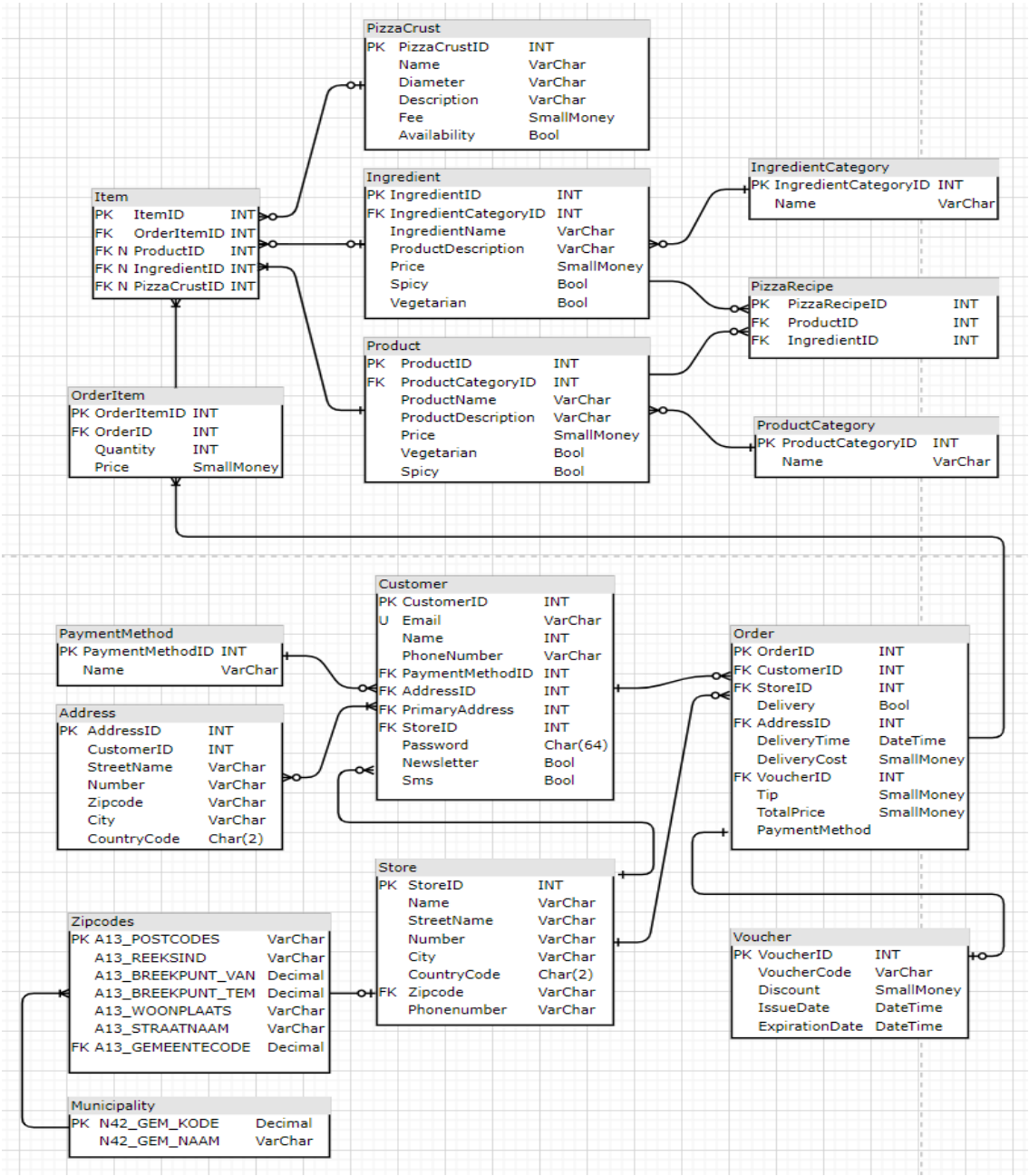

Fig. 3 - Connecting the tables

Connecting the tables with one another is the crucial thing. It's the deciding factor of the DB flexibility. We went for an ID Primary Key approach which is then mirrored as Foreign Key in other tables to make the connection. That way we couple tables together and don't have to rely on comparing data types other than simple Integer types. This makes using coupling tables more consistent with the overall design.

4. Final iteration

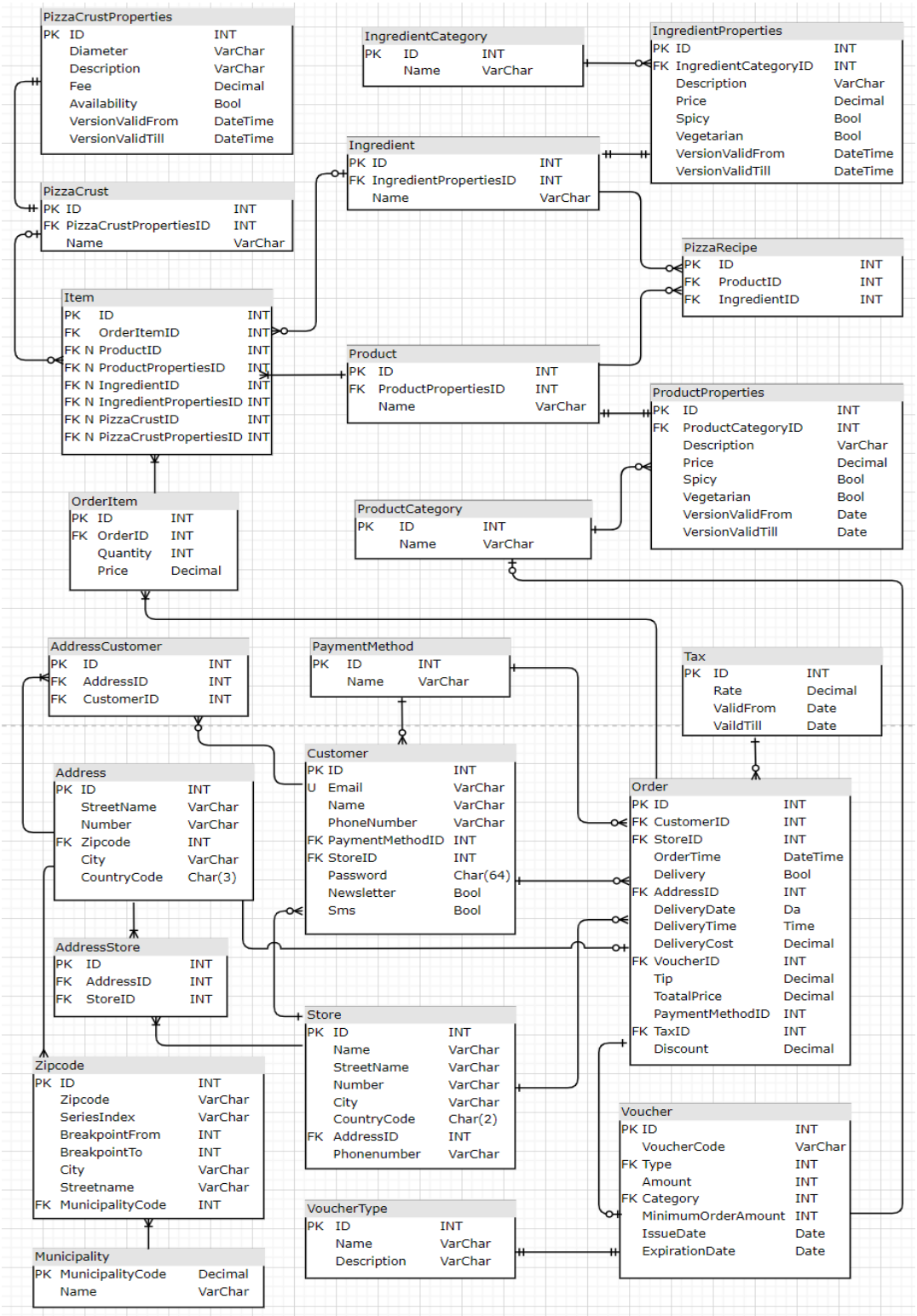

Fig. 4 - Final diagram

(description on p.9)

Description concerning Fig. 4 from p.8

SmallMoney type has been replaced by Decimal. This is due to the imprecision and possible calculation errors using this type of data which we found after researching the topic. We've decided to use following naming convention for best readability: "ID" name alone as PK; and FK in a "*SourceTable*ID" format. We use coupling tables to allow things like multiple Items in an Order or multiple addresses for customers and stores. This gives flexibility and future proofs the database making adding multiple addresses for customer possible. Properties tables store product qualities like descriptions, whether product is spicy or vegetarian, and whether product is valid or not. That way we can store products bought in the past in their original form and can accommodate things like product name changes without affecting archived orders. We have split the pizza ingredient's into it's own table, but included pizza's in the Product table. We think this gives better overview and fits the business requirements better. Pizza's are the core of the business, but at the same time they are a product just as any other. To discern between the variety of available products we use a Category table, same goes for Ingredient Category which discerns between ingredients on top of the pizza or sauces.
Tax table has been included to facilitate proper tax handling, which also includes ValidFrom/ValitTill fields to be able to specify time period in which a given tax rate has been applicable.
We think this data base design leaves room for future growth and fulfills the requirements set by existing MarioPizza business.

5.Helper diagram with data source:target routing

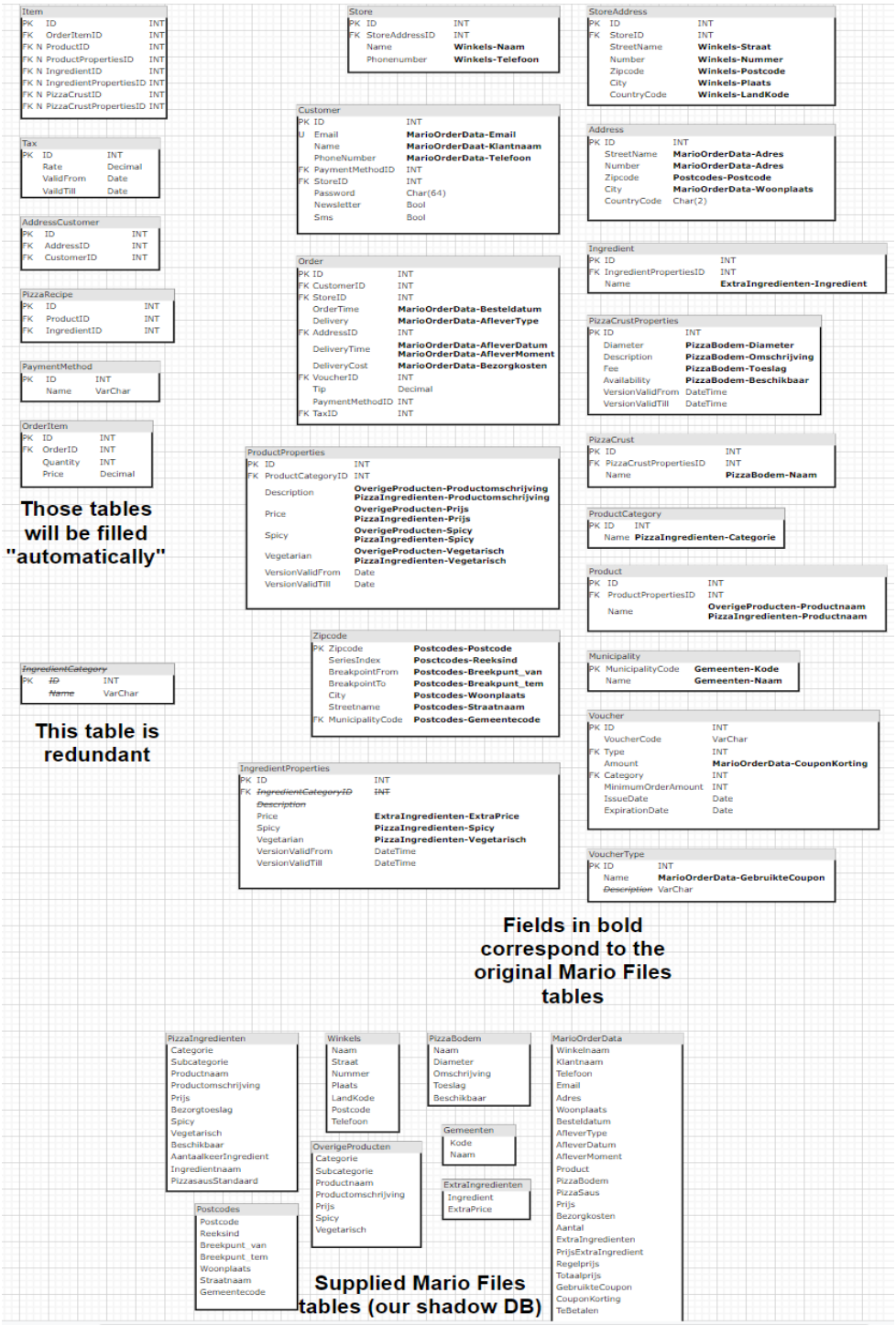

Fig. 5 - Helper diagram

Helper diagram used to determine which data goes where. We use it to illustrate the routing of the data that'll need to be transferred in our Stored Procedures queries. It's just a simple overview to keep trace of the data sources.

IMPLEMENTATION

For implementation we settled on Microsoft platform as it gives a coherent package to tackle the presented scenario. We use C# programming language to develop the software needed for cleaning-up and importing the supplied data as we are familiar with Java and C# seems similar in syntax. It has built in packages to connect to SQL Server which is our database of choice. Both C# and SQL Server have very good documentation and since they are very popular solutions there's a lot of instruction material available online.
Visual Studio IDE which we'll use for development, as well as SQL Server Management Studio are free for non-commercial use which makes it a perfect choice for the purpose of this project. We realize that for commercial use there are drawbacks too – like the pricing or the fact that SQL Server doesn't have an integrated version control capability (those are available from 3rd parties for additional charges).

Our approach to importing the data is to first copy all supplied data into a shadow DB as VARCHAR type. Since the shadow data base is just a temporary one, we are not concerned with any performance considerations at this point. During this process data will be cleaned by our software to get rid of any unexpected symbols, so that it's ready for further converting process into expected data types during the final import into the target Mario data base. Final import will be realized using Stored Procedures.
It seems to us that this approach offers us a good base and leaves room for any improvements/adjustments that might turn out necessary during the whole process. The use of Stored Procedures will hopefully allow us to limit our workload at further stages to SQL Server Management Studio and will make introducing changes more manageable.

As for the last step which would be visualizing the data we've settled on Power BI as our tool of choice. Again, it aligns with our needs, connects seamlessly to a broad variety of data sources (SQL Server included) and has a scope of functionality more than sufficient for our needs.

1. C# Mario Data Conversion Tool

```
class Order
{
    2 references
    public SqlString StoreName { get; private set; }
    2 references
    public SqlString CustomerName { get; private set; }
    2 references
    public SqlString PhoneNumber { get; private set; }
    2 references
    public SqlString Email { get; private set; }
    2 references
    public SqlString Address { get; private set; }
    2 references
    public SqlString City { get; private set; }
    2 references
    public SqlString OrderDate { get; private set; }
    2 references
    public SqlString DeliveryType { get; private set; }
    2 references
    public SqlString DeliveryDate { get; private set; }
    2 references
    public SqlString DeliveryTime { get; private set; }
    2 references
    public SqlString Product { get; private set; }
    2 references
    public SqlString PizzaCrust { get; private set; }
    2 references
    public SqlString PizzaSauce { get; private set; }
    2 references
    public SqlString Price { get; private set; }
    2 references
    public SqlString DeliveryCost { get; private set; }
    2 references
    public SqlString Amount { get; private set; }
    2 references
    public SqlString ExtraIngredients { get; private set; }
    2 references
    public SqlString ExtraIngredientPrice { get; private set; }
    2 references
    public SqlString LinePrice { get; private set; }
    2 references
    public SqlString TotalPrice { get; private set; }
    2 references
    public SqlString UsedVoucher { get; private set; }
    2 references
    public SqlString VoucherDiscount { get; private set; }
    2 references
    public SqlString TotalPriceAfterDiscount { get; private set; }
```

Fig. 6 - C# class mirroring supplied data (example)

We start by modelling the data supplied within the files. Each Class mirrors a table from the files ignoring data types.

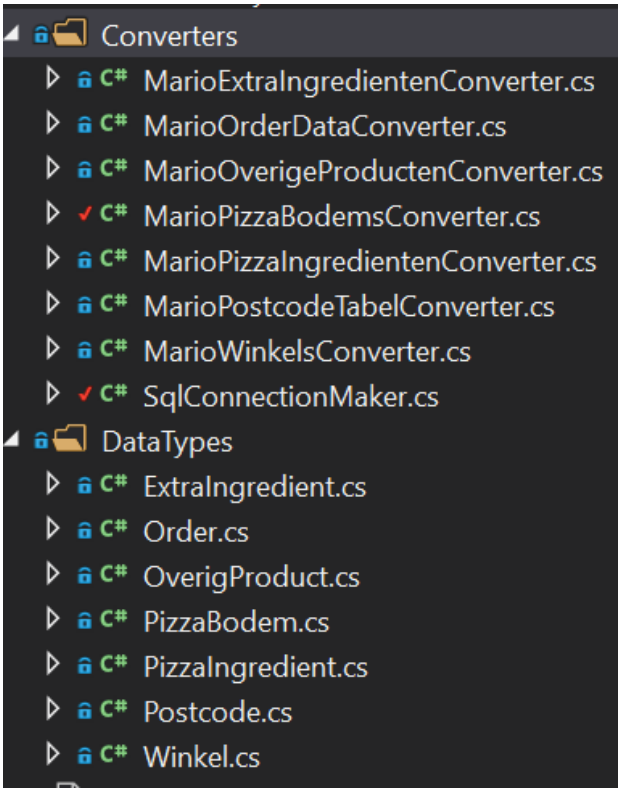

Fig. 7 - C# Classes overview

Each of the supplied files is being mirrored and each of them gets it's own Converter class which takes care of data cleaning and uploading into the shadow data base. The methods within each class are similar in structure. We will present one example from the PizzaBodems class to illustrate the approach we've taken.

```
class MarioPizzaBodemsConverter
{
    private static readonly log4net.ILog log = log4net.LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);
    private static readonly log4net.ILog logwarn = log4net.LogManager.GetLogger("logWarn");
```

Fig. 8 - C# Logger

Progress of each process is being logged by a logger. That way we can keep track of errors during the import process. All successful entries are logged into a log file. At the end of the standard log a summary is shows with the amount of lines read from each file. This ensures that all lines are read and no data is missing from the import.
All errors are being saved into a different file. The errors can be standard exceptions but also data that had to be changed because of discrepancies in the source files. The discrepancies are logged with remarks on the type of error, the file which contains the error and the line which contains the error. The customer can be notified of errors in the source material so that the source material can be corrected.
We decided to use *log4net* as our logging library as it is open source and offers all the functionality we could want. It has a reputation of being fast and reliable while remaining very flexible. There's a variety of methods of logging errors and events in your software and it can be configured at runtime. It is based on Java library *log4j*.

```
public List<PizzaBodem> ReadFile()
{
    log.Info("- - - - -");
    log.Info("Running : " + System.Reflection.MethodBase.GetCurrentMethod().Name);
    log.Info("- - - - -");

    ExcelPackage xlPackage = new ExcelPackage(new FileInfo(fileName));
    ExcelPackage.LicenseContext = LicenseContext.NonCommercial;

    string tempName = "";
    string tempDiameter = "";
    string tempDescription = "";
    string tempFee = "";
    Boolean tempAvailable;
    List<PizzaBodem> pizzaBodems = new List<PizzaBodem>();

    var myWorksheet = xlPackage.Workbook.Worksheets.First(); //select sheet here
    var totalRows = myWorksheet.Dimension.End.Row;
    var totalColumns = myWorksheet.Dimension.End.Column;

    for (int rowNum = 2; rowNum <= totalRows; rowNum++) //select starting row here
    {
        tempName = myWorksheet.GetValue(rowNum, 1).ToString();
        tempDiameter = myWorksheet.GetValue(rowNum, 2).ToString();
        tempDescription = myWorksheet.GetValue(rowNum, 3).ToString();
        tempFee = myWorksheet.GetValue(rowNum, 4).ToString();
        tempFee = new string(tempFee.Where(c => (Char.IsDigit(c)||c =='.'||c==',')).ToArray());
        tempFee = tempFee.Replace(",", ".");
        if (myWorksheet.GetValue(rowNum, 5).ToString() == "Ja")
        {
            tempAvailable = true;
        } else
        {
            tempAvailable = false;
        }

        pizzaBodems.Add(new PizzaBodem(tempName, tempDiameter, tempDescription,
            decimal.Parse(tempFee, CultureInfo.InvariantCulture), tempAvailable));
        log.Info("Succesfully added line:" + rowNum);
        tempName = "";
        tempDiameter = "";
        tempDescription = "";
        tempFee = "";
        tempAvailable = false;

    }
    return pizzaBodems;
}
```

Fig. 9 - C# Filling item list

Items from original Mario files are instantiated as objects based on prepared classes and are loaded into a list. At this stage we loop through the rows and convert them to Strings, replace commas with period signs to prepare the data for conversion into numeric format later on and prepare the Boolean data for further conversion by setting them to *true/false* values following a more prevalent programming convention. On each iteration of the loop an object is being created and added to the list.

Above an Excel file (*.xlsx) is being loaded hence the use of *ExcelPackage* from the *EPPlus* library. This library has no external dependencies when working within the .NET environment and is free for non-commercial use. For other file types we use the following: *SystemIOStreamReader* for *.csv and *.txt files; *OleDBDataReader* via *OleDBConnection* to connect to MS Access *.mdb files. Both available as standard .NET environment offering basic functionality for the needs of the project.

```
public void Upload(List<PizzaBodem> pizzaBodems)
{
    log.Info("- - - - -");
    log.Info("Running : " + System.Reflection.MethodBase.GetCurrentMethod().Name);
    log.Info("- - - - -");

    //Drop table and create new one
    SqlConnection conn = SqlConnectionMaker.ReturnConnection();

    try
    {

        conn.Open();
        SqlCommand command = conn.CreateCommand();
        command.CommandText = "DROP TABLE IF EXISTS dbo.PizzaBodems";
        command.ExecuteNonQuery();
        command.CommandText = "CREATE TABLE dbo.PizzaBodems (Naam varchar(200),Diameter varchar(200)," +
            "Omschrijving varchar(200),Toeslag varchar(200),Beschikbaar varchar(200))";
        command.ExecuteNonQuery();


    }
    catch (Exception)
    {
        throw;
    }
    finally
    {
        conn.Dispose();
        conn.Close();
    }

    foreach (PizzaBodem pizzaBodem in pizzaBodems)
    {

        ExecuteQuery(pizzaBodem.Name.ToString(),
            pizzaBodem.Diameter.ToString(),
            pizzaBodem.Description.ToString(),
            pizzaBodem.Fee.ToString(),
            pizzaBodem.Available.ToString());
    }

}
```

Fig. 10 - C# Upload

First we establish a connection to the right SQL instance using a username and password. These settings can be altered in the SqlConnectionMaker.cs file. After establishing a connection with the data base we execute queries to first drop the table if it exists and then create a new one. That way we wanted to guarantee data integrity. Connection and queries execution are contained within a *Try/Catch* block to catch exceptions and after execution we close the connection. A *for* loop executes the *INESRT* query filling the rows on each iteration of the loop.

```
private void ExecuteQuery(
    string Name,
    string Diameter,
    string Description,
    string Fee,
    string Available)
{
    SqlConnection conn = SqlConnectionMaker.ReturnConnection();

    String query = "INSERT INTO dbo.PizzaBodems (Naam,Diameter,Omschrijving," +
        "Toeslag,Beschikbaar) " +
        "VALUES (@Naam,@Diameter,@Omschrijving,@Toeslag,@Beschikbaar)";
    try
    {

        conn.Open();
        SqlCommand command = conn.CreateCommand();
        command.CommandText = query;

        command.Parameters.AddWithValue("@Naam ", Name);
        command.Parameters.AddWithValue("@Diameter ", Diameter);
        command.Parameters.AddWithValue("@Omschrijving ", Description);
        command.Parameters.AddWithValue("@Toeslag ", Fee);
        command.Parameters.AddWithValue("@Beschikbaar ", Available);

        command.ExecuteNonQuery();
    }
    catch (Exception)
    {
        throw;
    }
    finally
    {
        conn.Dispose();
        conn.Close();
    }
}
```

Fig. 11 - C# Insert query execution

This is the *INSERT* query execution method. Actual values are being passed as arguments to the AddWithValue function.

2. Import - SQL Stored Procedures

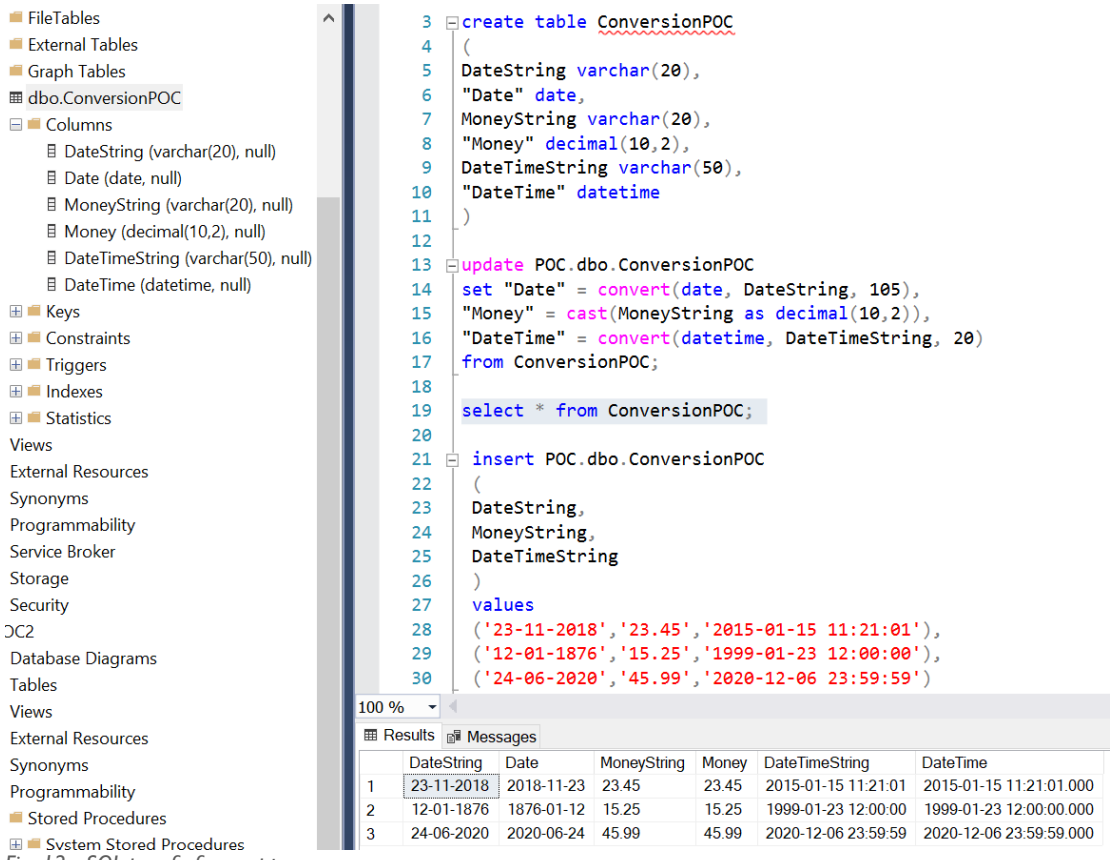

Fig. 12 - SQL proof of concept

As we've decided to do the data type conversion withing the SQL Server environment we start by creating proof of concepts snippets to test the conversion process. This proved to be straightforward as long as the supplied data is clean. That's why we make sure to clean the data during import into the shadow DB and to use proper formatting for date and time fields.

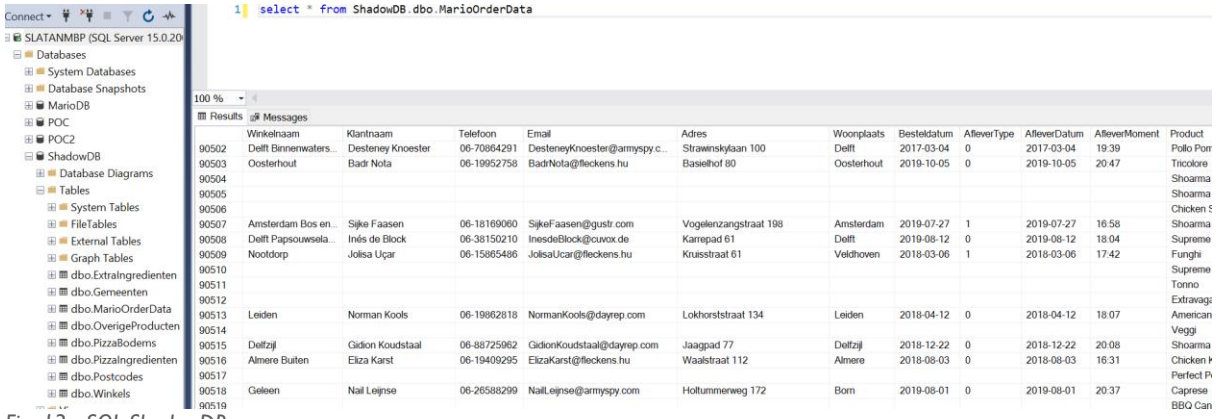

Fig. 13 - SQL ShadowDB

MarioOrderData as viewed in our ShadowDB. It basically mirrors the original file with some improvements to formatting and cleaning up of the unnecessary empty rows etc.

```
ALTER PROCEDURE spImportIntoMarioDB
AS
BEGIN
SET IDENTITY_INSERT Municipality ON
--SET IDENTITY_INSERT Zipcode ON
-----------------------------------------------------
        INSERT INTO dbo.Municipality
        (
        MunicipalityCode,
        "Name"
        )
        SELECT Kode, Naam
        FROM ShadowDB.dbo.Gemeenten
-----------------------------------------------------
        INSERT INTO dbo.Zipcode
        (
        Zipcode,
        SeriesIndex,
        BreakpointFrom,
        BreakpointTo,
        City,
        Streetname,
        MunicipalityCode
        )
        SELECT Postcode, Reeksind, Breekpunt_van, Breekpunt_tem, Woonplaats,Straatnaam,Gemeentecode
        FROM ShadowDB.dbo.Postcodes
-----------------------------------------------------
        INSERT INTO dbo.VoucherType
        (
        "Name"
        )
        SELECT GebruikteCoupon
        FROM ShadowDB.dbo.MarioOrderData
-----------------------------------------------------
        INSERT INTO dbo.ProductCategory
        (
        "Name"
        )
        SELECT Categorie
        FROM ShadowDB.dbo.PizzaIngredienten
-----------------------------------------------------
```

Fig. 14 - SQL 1st draft INSERT SP

As our data are placed in a data base already, we start to put together our INSERT stored procedures. On the supplied screenshot pictured a very 1st simple draft that copies the data between the two data bases. This version of the SP is far from finished however since it doesn't properly redistribute the data into its target locations.

```
-------------------------------------------------
INSERT INTO Ingredient
(
"Name"
)
SELECT Ingredient
FROM ShadowDB.dbo.ExtraIngredienten
-------------------------------------------------
INSERT INTO Customer
(
Email,
"Name",
PhoneNumber
)
SELECT Email, Klantnaam, Telefoon
FROM ShadowDB.dbo.MarioOrderData WHERE Email != ShadowDB.dbo.MarioOrderData.Email
-------------------------------------------------
INSERT INTO "Order"
(
OrderTime,
Delivery,
DeliveryDate,
--DeliveryTime,
DeliveryCost
)
SELECT Besteldatum, /*= convert(date, Besteldatum, 3),*/
CASE WHEN AfleverType = '0' THEN CAST('0' as bit) ELSE CAST('1' as bit) END,
AfleverDatum /*= convert(date, AfleverDatum, 3)*/,
--AfleverMoment = try_convert(time(0), AfleverMoment, 0),
Bezorgkosten = try_parse(Bezorgkosten as decimal(10,2) USING 'El-GR')
FROM ShadowDB.dbo.MarioOrderData
```

Fig. 15 - SQL further SP development

A further improvement on the SP which now converts the data to the required types. Converting to Boolean type can be seen in the *CASE* statement. Some issues we've had is proper date/time string formatting due to the limited possibilities of formatting within SQL Server. We managed to solve it by pre-formatting in the C# software, so the string matches the required format.

```
INSERT INTO "Address"
(
Zipcode
)
SELECT ID =

CASE
WHEN (cast((select HouseNumber from "Address") as int) % 2 <> 0)
    THEN
    (
    SELECT ID FROM dbo.Zipcode
    WHERE
    (
    (City = (select City from dbo.Zipcode) AND Streetname = (select Streetname from dbo.Zipcode))
    AND (dbo.Zipcode.BreakpointFrom % 2 <> 0
    AND (cast((select HouseNumber from "Address") as int) >= (select BreakpointFrom from Zipcode)
    AND cast((select HouseNumber from "Address") as int) <= (select BreakpointTo from Zipcode)))
    )
    )
ELSE
    (
    SELECT ID FROM dbo.Zipcode
    WHERE
    (
    (City = (select City from dbo.Zipcode) AND Streetname = (select Streetname from dbo.Zipcode))
    AND (dbo.Zipcode.BreakpointFrom % 2 = 0
    AND (cast((select HouseNumber from "Address") as int) >= (select BreakpointFrom from Zipcode)
    AND cast((select HouseNumber from "Address") as int) <= (select BreakpointTo from Zipcode)))
    )
    )
END
FROM dbo.Zipcode
```

Fig. 16 - SQL housenumber zipcode matching

The zipcode table contains the street name and ranges of house numbers – separate for odd and separate for even numbers. An address has to be not only within a given range, but it also has be compared against the right range – odd or even. To couple the proper zipcode to given address we decided on using *MODULO* to identify whether house number is odd or even and then to compare whether it's within the given range. *MOD* is also used to identify whether a row concerns even or odd ranges. This early draft doesn't work properly.

```
INSERT INTO StoreAddress
    (
    Streetname,
    Number,
    Zipcode,
    City,
    CountryCode
    )
    SELECT DISTINCT Straat, Nummer, z.ID, UPPER(Plaats), Landcode
    FROM ShadowDB.dbo.Winkels w, MarioDB.dbo.Zipcode z
    WHERE w.Postcode = replace(z.Zipcode,' ','') AND
    ((dbo.udf_GetNumeric(Nummer) Between z.BreakpointFrom And z.BreakpointTo
       Or dbo.udf_GetNumeric(Nummer) Between z.BreakpointTo And z.BreakpointFrom)
       And (dbo.udf_GetNumeric(Nummer) % 2 = z.BreakpointFrom % 2)) AND
       NOT EXISTS (SELECT * FROM StoreAddress sa
    WHERE sa.Streetname = Straat and sa.City = UPPER(Plaats) and sa.Number = Nummer)
```

Fig. 17 - SQL housenumber zipcode matching 2

This is the working version of the SP from previous page. We've included a check whether given data exists in the DB already to avoid duplicates. Working on this SP it turned out not all zip codes are included within the supplied files making binding many of the addresses from supplied files impossible. This would have to be reported back to the customer and a solution would have to be pursued.

- dbo.sp01ImportMarioMunicipality
- dbo.sp02ImportMarioZipcode
- dbo.sp03ImportMarioVoucherType
- dbo.sp04ImportMarioProductCategory
- dbo.sp05ImportMarioVoucher
- dbo.sp06ImportMarioPizzaCrustProperties
- dbo.sp07ImportMarioPizzaCrust
- dbo.sp08ImportMarioIngredientCategory
- dbo.sp08ImportMarioIngredientProperties
- dbo.sp09ImportMarioIngredient
- dbo.sp10ImportMarioCustomer
- dbo.sp11ImportMarioProductCategory
- dbo.sp12ImportMarioProductProperties
- dbo.sp13ImportMarioProduct
- dbo.sp14ImportMarioPizzaRecipe
- dbo.sp15ImportMarioTax
- dbo.sp16ImportMarioPaymentMethod
- dbo.sp23ImportOLDMarioAddress
- dbo.sp24ImportMarioStoreAddress
- dbo.sp25ImportMarioStore
- dbo.sp30Import!MarioOrder
- dbo.spExecuteProcedures
- dbo.spImportIntoMarioDB
- dbo.spReport1StoreAddressNotImported

Fig. 18 - SQL all stored procedures used

```
CREATE PROCEDURE [dbo].[spExecuteProcedures]
AS

DECLARE @SPECIFIC_SCHEMA varchar(200), @SPECIFIC_NAME varchar(200)

SELECT SPECIFIC_SCHEMA, SPECIFIC_NAME
INTO #tempStoredProcedures
    FROM MarioDB.INFORMATION_SCHEMA.ROUTINES
    WHERE ROUTINE_TYPE = 'PROCEDURE' AND ROUTINE_NAME LIKE '%ImportMario%'
    ORDER BY ROUTINE_NAME ASC

WHILE EXISTS(SELECT * From #tempStoredProcedures)
BEGIN
    SET NOCOUNT ON;

    Select Top 1 @SPECIFIC_SCHEMA = SPECIFIC_SCHEMA, @SPECIFIC_NAME = SPECIFIC_NAME
    From #tempStoredProcedures

    EXECUTE (@SPECIFIC_SCHEMA + '.' + @SPECIFIC_NAME);

    Delete #tempStoredProcedures Where SPECIFIC_NAME = @SPECIFIC_NAME
END

DROP TABLE #tempStoredProcedures

GO
/****** Object:  StoredProcedure [dbo].[spImportIntoMarioDB]    Script Date: 15-10-2020 11:05:32 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[spImportIntoMarioDB]
AS
BEGIN
```

Fig. 19 – SQL Execute all stored procedures

We have contained all of our Stored Procedures within a script, which is a functionality within the SQL Server Management Studio. The creation of the data base itself, all PK/FK constraints are also handled by the script. This allows us to quickly set the whole data base back up again when the data gets too messy due to the trial and error during testing. After running the script we have all the SP's already created and it's just a matter of running them all in consecutive order. This is being handled by the pictured SP. It looks for specific naming scheme and executes the SP's till it reaches the last one, at which point the execution is terminated.

3. Visualization

For the visualization step we chose to use Power BI Desktop. It supports many different types of data sources including SQL Server obviously. It supports Stored Procedures execution including *sp_execute_external_script* which allows running *Python* scripts for example. Moreover it can either work in DirectQuery mode which sends queries, as the name suggests, directly to SQL Server each time there's a visualization to be displayed or in Import mode which imports all the data in question into the memory. Mainly for performance reasons we'll go with the latter approach. DirectQuery mode requires the database to be well optimized especially where large data sets are concerned. Otherwise, we'd have to deal with waiting times during presentations which is an inconvenience, we feel, one doesn't want to be faced with during a business meeting. From our research we gathered that data base optimization is an expansive topic, well beyond the scope of this assignment. This kind of direct approach has its advantages too – data doesn't need to be refreshed each time new records are added, since the visualization is a result of pulling the information in from the data base itself. In broad terms one could say the DirectQuery is a live/online approach and Import is similar to an offline mode and needs refreshing each time the information has been updated/changed.

How many different pizza sorts have been sold?

```
select sum(oi.Quantity) as "Pizzas Sold",
pc."Name" as “Pizza Category”, p.”Name” as “Pizza Name”,
YEAR(o.DeliveryDate) as "Year" from Product p
join Item i on i.ProductID = p.ID
join OrderItem oi on i.OrderItemID = oi.ID
join "Order" o on oi.OrderID = o.ID
join ProductProperties pp on p.ProductPropertiesID = pp.ID
join ProductCategory pc on pp.ProductCategoryID = pc.ID
join Product p on p.ID = i.ProductID
where (pc.ID between 1 and 3)
group by pc."Name", YEAR(o.DeliveryDate)
```

What are the worse selling products excluding pizzas per year?

```
select sum(oi.Quantity) as "Sold", pc."Name" as ProductCategory,
YEAR(o.DeliveryDate) as "Year" from Product p
join Item i on i.ProductID = p.ID
join OrderItem oi on i.OrderItemID = oi.ID
join "Order" o on oi.OrderID = o.ID
join ProductProperties pp on p.ProductPropertiesID = pp.ID
join ProductCategory pc on pp.ProductCategoryID = pc.ID
where (pc.ID between 4 and 20)
group by pc."Name", YEAR(o.DeliveryDate)
```

Which shops get the best/worse turnover over the years?

```
select sum(o.Totalprice) as Turnover,s."Name",
from "Order" o
join Store s on o.StoreID = s.ID
group by s.Name, Turnover order by s."Name" desc
```

Which are clients favourite pizza?

```
select Client, Pizza, Bought
from
(
select c."Name" as Client, p.Name as Pizza, sum(oi.Quantity) as Bought,
row_number() over (partition by c.Name order by sum(oi.Quantity) desc) as rn
 from Customer c
join "Order" o on o.CustomerID = c.ID
join OrderItem oi on oi.OrderID = o.ID
join Item i on i.OrderItemID = oi.OrderID
join Product p on p.ID = i.ProductID
join ProductProperties pp on p.ProductPropertiesID = pp.ID
where pp.ProductCategoryID between 1 and 3
group by c.Name, p.Name
) as Temp
where rn <= 1
order by Bought desc
```

TOP 3 pizzas per Store?

```
Select "Store Name", Pizza, Sold
from (
select
s.Name as "Store Name",
p.Name as Pizza,
sum(oi.Quantity) as Sold,
ROW_NUMBER() over (partition by s.Name order by sum(oi.Quantity) desc) as rn
from Store s
join "Order" o on o.StoreID = s.ID
join OrderItem oi on oi.OrderID = o.ID
join Item i on i.OrderItemID = oi.ID
join Product p on p.ProductPropertiesID = i.ProductPropertiesID
join ProductProperties pp on p.ProductPropertiesID = pp.ID
where pp.ProductCategoryID between 1 and 3
group by p.Name, s.Name
) as Temp
where rn <= 3;
```

Fig. 20 PowerBI - business questions and queries

We start by specifying some business-related questions concerning the Mario data and simultaneously by writing accompanying queries to be used in PowerBI. We do this for efficiency reasons - we can use *T-SQL* queries during data import for visualization purposes, limiting the amount of data imported to the specific data we used for any given chart.

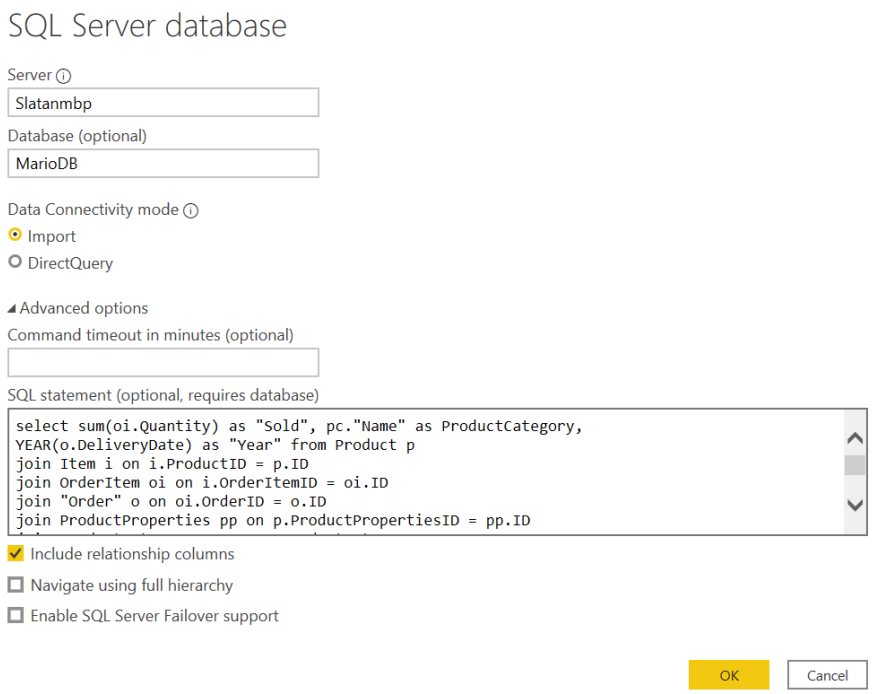

Fig. 21 PowerBI importing data specified by the query

After importing the limited set of data into Power BI we are still able to filter out information using the Power BI built in filters.

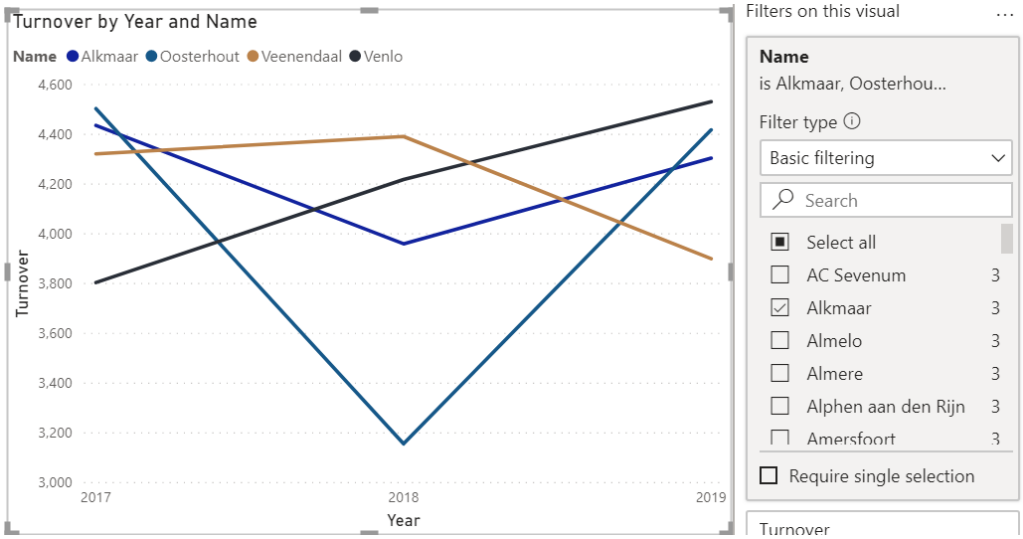

Fig. 22 PowerBI built in filtering options