UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

ZAKLJUČNA NALOGA
(FINAL PROJECT PAPER)

ANALIZA IN PRIMERJAVA NYC TAXITRIP 2019/2020 NABOROV PODATKOV Z UPORABO SQL STREŽNIKA IN POWER BI

(ANALYSIS AND COMPARISON OF NYC TAXITRIP DATASETS 2019/2020 WITH SQL SERVER AND POWER BI)

ELMIR ŠUT

2021
ZAKLJUČNA NALOGA (FINAL PROJECT PAPER)
ŠUT

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga
(Final project paper)

Analiza in primerjava NYC Taxitrip 2019/2020 naborov podatkov z uporabo SQL strežnika in Power BI

(Analysis and comparison of NYC Taxitrip datasets 2019/2020 with SQL server and Power BI)

Ime in priimek: Elmir Šut
Študijski program: Računalništvo in informatika
Mentor: doc. dr. Uroš Godnov

Koper, november 2021

Ključna dokumentacijska informacija

Ime in PRIIMEK: Elmir ŠUT

Naslov zaključne naloge: Analiza in primerjava NYC Taxitrip 2019/2020 naborov podatkov z uporabo SQL strežnika in Power BI

Kraj: Koper

Leto: 2021

Število listov: 62 Število slik: 61 Število tabel: 1

Število referenc: 17

Mentor: doc. dr. Uroš Godnov

Ključne besede: množični podatki, tabele, obdelava, SQL strežnik, Power BI, podatkovna analiza, poročila

Izvleček:

V tej zaključni nalogi se analizirajo in primerjajo podatki o prevozih taksijev, zbrani v letih 2019 in 2020 v New Yorku, tako da je mogoče sklepati o tem, kako je svetovna pandemija Covid-19 vplivala na vsakodnevno uporabo taksijev kot prevoznega sredstva. Trenutno ni izvedenih toliko študij in poročil na podlagi tega javnega nabora podatkov v kontekstu razlik med podatki, zbranimi v času pred pandemijo (celotno leto 2019, prva dva meseca leta 2020) in v času pandemije (zadnjih 10 mesecev leta 2020). Glavni cilj zaključne naloge je ugotoviti, kako in v kolikšni meri je bila prizadeta industrija taksi prevoza z javnimi zdravstvenimi in socialnimi ukrepi za boj proti COVID-19 pandemiji, ter kako je na vedenje strank vplivala pandemija glede na vrsto plačila in število strank, ki si delijo prevoz. Zaključna naloga bo opisala tudi uporabo SQL strežnika, integracijske storitve SQL strežnika v smislu uvoza, čiščenja in shranjevanja podatkov ter nastavljanja omejitev in particij. Opisana bo tudi uporaba Power BI kot orodja za analizo in interaktivno vizualizacijo podatkov. Končno bo prikazano vizualizirano poročilo na podlagi analiz in bo pomagalo kvantificirati, kako je pandemija spremenila vsakodnevne dejavnosti, povezane z uporabo taksi storitev.

Key document information

Name and SURNAME: Elmir ŠUT

Title of the final project paper: Analysis and comparison of NYC Taxitrip datasets 2019/2020 with SQL Server and Power BI

Place: Koper

Year: 2021

Number of pages: 62 Number of figures: 61 Number of tables: 1

Number of references: 17

Mentor: Assist. Prof. Uroš Godnov, PhD

Keywords: Big Data, tables, manipulation, SQL Server, Power BI, data analysis, reports

Abstract:

In this final project paper, Taxi record data collected in 2019 and 2020 in New York City is analyzed and compared, so that conclusions can be drawn on how the global pandemic of Covid-19 affected the everyday use of taxis as a means of transportation. There are no currently so many studies conducted and reports produced on this public dataset in the context of differences between data collected during pre-pandemic (entire 2019, first two months of 2020) and peri-pandemic (last 10 months of 2020). The main goal of this final project paper is to identify how and to what extent the taxi transportation industry was affected by public health and social measures for COVID-19, how the behavior of customers was affected in terms of payments and the number of customers sharing the transportation. Also, how it affected average traveling time. The paper will also describe the usage of SQL Server, SQL Integration Services in terms of data import, cleaning, storage, and setting constraints and partitions. Power BI usage as analytics and interactive data visualization tool will be described, too. Finally, it will show the visualized report based on analyses and help quantify how pandemics changed day-to-day activities related to the usage of Taxi services.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my mentor and Assistant Prof. Uroš Godnov for his support during the idea development and shaping process and guidance throughout the working of this final project paper.
I thank my supervisors at my job Gasper Žerak and Robert Buk for the support in terms of knowledge and resources.
I thank the Faculty of Mathematics, Natural Sciences, and Information Technologies for giving me opportunities to learn and implement many current trends in computer and data science.
Finally, I would like to thank my family for their enormous support during my studies in the Republic of Slovenia.

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVATIONS

etc.	and other similar things
SQL	Structured Query Language
TSQL	Transact-SQL
SSIS	SQL Server Integration Services
TLC	Taxi & Limousine Commission
SSMS	SQL Server Management Studio
RDBMS	Relational Database Management System
NYC	New York City
CSV	Comma Separated Value
USD	United States dollar
DAX	Data Analysis Expressions language

1 INTRODUCTION

As my undergraduate studies were coming closer to their end, I started to think about how the choice of my final thesis topic could shape my future career in computer science. I always wanted to make something practical and sought after in the market. For that reason, I contacted many companies in the field for advice. My final decision, based on the feedback I got, was to focus on data collection, manipulation, and Big Data in general. Furthermore, emphasis was placed on business intelligence platforms like Power BI. The expression "Big data" usually refers to large amounts of data impossible to even be imported to tools like Microsoft Excel or Google Sheets. Big Data Analytics as a process encompasses all of aforementioned data analysis components and many more performed on huge amounts of data in order to find patterns that will help draw conclusions and provide meaningful insights. Big Data and Power BI come together as long as data is structured, which means that it is defined in terms of tables consisting of rows and columns with established relations and connections. I had also successfully completed three elective courses in the third year of studies regarding the use of Structured Query Language (SQL), SQL Server, and data visualization. SQL is mostly used to share and manage data in relational database management systems (RDBMS). Data in RDBMS is organized into tables. Tables may be related together by a common field. By using SQL, it is possible to query data. Also, it is possible to update and reorganize data, modify a database, or limit access to data. All of the three subjects were taught by Assistant Professor Uroš Godnov. They were extraordinarily practical and pragmatic in all respects. Having all of those experiences in my CV helped me to get my first job as a Business Intelligence/Datawarehouse consultant in a company operating in the DACH region (Germany, Austria, Switzerland). After seven months of work on various international projects, some of which assigned to a large extent to me, I have gained enough academic courage to propose my intention to do a data science project and research as my final thesis. My first and only choice was Assistant Professor Godnov. Luckily, he responded positively even though at the time due to his overloaded schedule he was not teaching those subjects on our faculty. A few days later we had our first video call meeting, where he already had a project in his mind, based on my preferences and under his mentorship. He informed me about the aims of the proposed project, the data that needs to be processed, what tools should be used and in what manner. According to his input, a list of hypotheses was (see **subchapter 1.1**) that will be evaluated in the final part of the project. So, the idea was to use large amount of taxi trip public data collected during 2019 and 2020. Choosing NYC taxi trip datasets as the starting point was an easy decision due to the global pandemic of coronavirus disease 2019 (COVID-19) hitting New York City hard, as well as the rest of the world. The first patient infected with the COVID-19 coronavirus in New York City was detected on the 1st of March 2020 (see [2] for more information). Only

40 days after the first patient got infected with COVID-19, New York had more confirmed cases than any country in the world excluding the US (see [3] for further information). The Data was originally stored in separate files, so the aim was to merge all data files in one, set data criteria, load it in a table and use the table to create an interactive report. In the report, hypotheses should be proved or disproved based on overall taxi trips recorded, the average time taken in a single trip, the count of payments made according to its type, and average passenger counts in a single trip during pre-pandemic months compared to peri-pandemic months.

1.1 Hypotheses

The main purpose of this final thesis is to analyze and compare large amounts of Taxi record data collected in the years 2019 and 2020 in New York City, so that it is understandable how global pandemic of Covid-19 affected everyday use of taxis as a means of transportation. So far, not many studies have been conducted or reports produced on this public dataset in a context of differences between data collected during the pre-pandemic (entire 2019, first two months of 2020) and during the peri-pandemic (last 10 months of 2020). The main goal of this final thesis is to identify how and to what extent the taxi transportation industry was affected by public health and social measures for COVID-19, and how the behavior of customers was affected in terms of travel time, payments, and number of customers sharing the transportation.

Hypotheses:

1. There is significant difference in the amounts of taxi trips recorded during pre-pandemic months compared to peri-pandemic months.
2. There is significant difference in average time taken to arrive from pick-up to drop-off location during pre-pandemic months compared to peri-pandemic months in relation to trip length.
3. There is significant difference in the amounts of cash payments compared to credit card payments during pre-pandemic and peri-pandemic months.
4. There is significant difference in the amounts of driver-reported passenger counts in a single trip during pre-pandemic months compared to peri-pandemic months.

1.2 Data

Data was available publicly at the official The New York City Taxi and Limousine Commission (TLC) website (see [1]). For each month there are two CSV (Comma separated value) files with collected data. In total, there are 48 CSV files. There are two categories of taxies in New York City: yellow taxies and green taxies. The difference is

that green taxis can pick up passengers only in the outer parts of New York City (see [4] for more info). There are green and yellow taxi trip data CSV files for each month of 2019/2020. The yellow and green taxi trip records contain fields that hold the dates and times when a passenger was picked up and dropped off, locations of pick-up, location of drop-off, trip distances, detailed charges, rate categories, payment categories, and driver-reported passenger counts (for more info see [1]).

1.3 Installations

Advantageously, all tools needed for the project were previously installed and set up except for *Microsoft Power BI*. Those tools are *Microsoft Visual Studio Community 2019* (Version 16.7.4), SQL *Server Integration Services Project* extension for *Microsoft Visual Studio Community 2019*, SQL *Server 2019 Developer edition 64-bit, SQL Server Management Studio 2019.* The only tool needed to be installed was *Microsoft Power BI – Desktop Edition (*see download link [5]). All of the tools used are developed by Microsoft.

2 SQL SERVER

SQL Server (Microsoft SQL Server) is a Relational Database Management System. It is a piece of software that stores data (database) and is capable of executing SQL commands and queries. Commands that manipulate the relational database. Furthermore, SQL Server manages and performs all database processes (for more info on SQL Servers see [7]).
SQL Server Integration Service (SSIS) is a part of the Microsoft SQL Server database software generally used to execute a broad range of data migration and manipulation tasks. SSIS is a straightforward and reliable tool used for data extraction, loading, and transformations such as fixing incorrect data, removing duplicates, combining more datasets, converting data, storing data, etc. (for more info see [6]). SSIS is available as an extension of Visual Studio. SSIS enables the user to modularize workload into packages. Every package in SSIS is a .DTSX file, meaning that it is an XML-based file format that can be created and used by SQL Server software. Each package can consist of one or many *Control Flow Tasks* and each *Control Flow Task* may have underlying *Data Flow components (sources, destinations, functions, etc.).*
In a *Control Flow Task, the* smallest unit of work is the task. The order in which tasks must be executed is specified by precedence constraints (*green, red, and grey connectors, which specify the flow order at run time*). Precedence constraints are also used for managing the workflow of *Data Flow* components.

2.1 Preparation of unified CSV file

Before importing the data to the SQL Server, certain actions to ensure data consistency are to be performed. All 48 large CSV files need to be saved in one dedicated map. This part was done manually. Afterward, the CSV files need to be merged into one unified CSV file. This makes it easier to clean and manipulate data all at once. It assures that all files follow the needed structure. If an extra column needs to removed or added, it should be done just once when importing the unified CSV file in the database (more in **subchapter 2.3**). Also, if columns need to be renamed, it would be done in just one place. The new CSV file is a flat-file source for the database. All the data from the aforementioned 48 large CSV files is also present in the new unified CSV file.
To perform all of these actions systematically and to also make them consistent, SSIS is used. It is important to emphasize that if any SSIS packages are run several times, data must be consistent each time. Old data should be emptied and the new data loaded. To prepare a directory for the future *unified CSV file*, to make sure it is always empty before loading the data into it, and to union all files with the right formatting, an SSIS package named *CSVFilePreparation* is created. In the package *CSVFilePreparation.dtsx,* the data is prepared and stored in the unified CSV file. The main part of the work happens in the Data

Flow of this package's final Control Flow Task. Every part of the package is explained in detail below. The structure of the abovementioned package is visible in **Figure 1**.

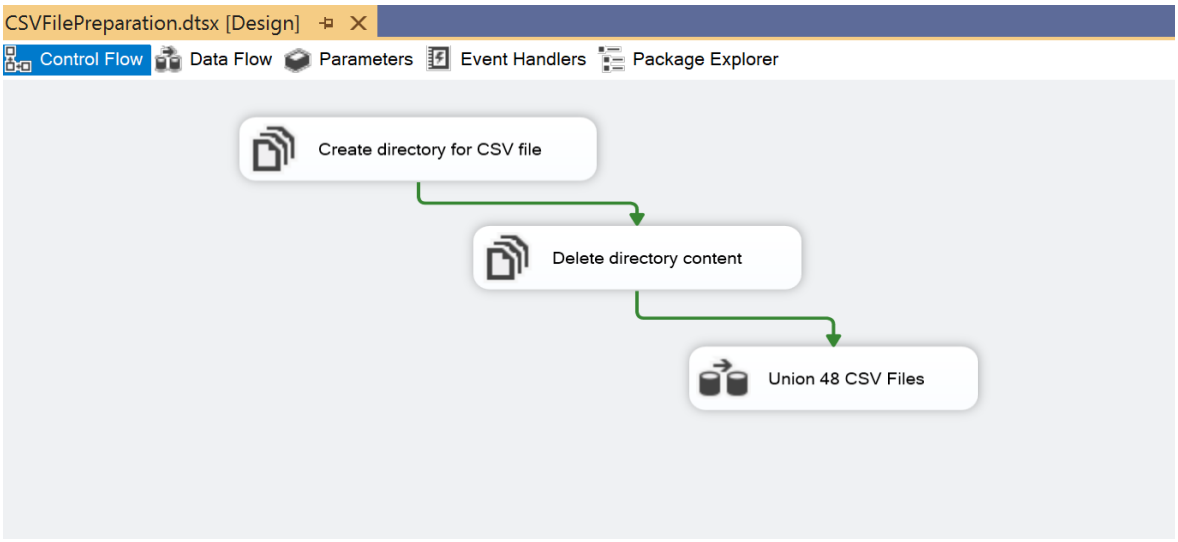


Figure 1: *CSVFilePreparation.dtsx* package structure

2.1.1 Creation of directory for the CSV file

To prepare a proper directory for the future CSV file, a *File System Task* named *Create directory for CSV file is used* (see **Figure 1**). The task is able to perform many different operations on files and directories. It can be used to create a directory, copy a directory, delete a directory, delete directory content, move a directory, copy a file, delete a file, move a file, or set attributes. In this specific *Control Flow task, the create directory* option is used. The newly created directory will hold the CSV file that will be created during run-time by the Data flow *Union all CSV files into one* task (more on this in **section 2.1.3**). In the *File System Task Editor*, shown in **Figure 2**, the option *true* is selected. That way, in the case the directory specified by the user-defined source variable already exists, that directory will be used instead. If the option false is selected and the directory already exists, the task would fail, resulting in the failure of the package itself. In the case that data already exists in the directory it would be deleted, but it will be dealt with in another *file system task* (see **section 2.1.2**).

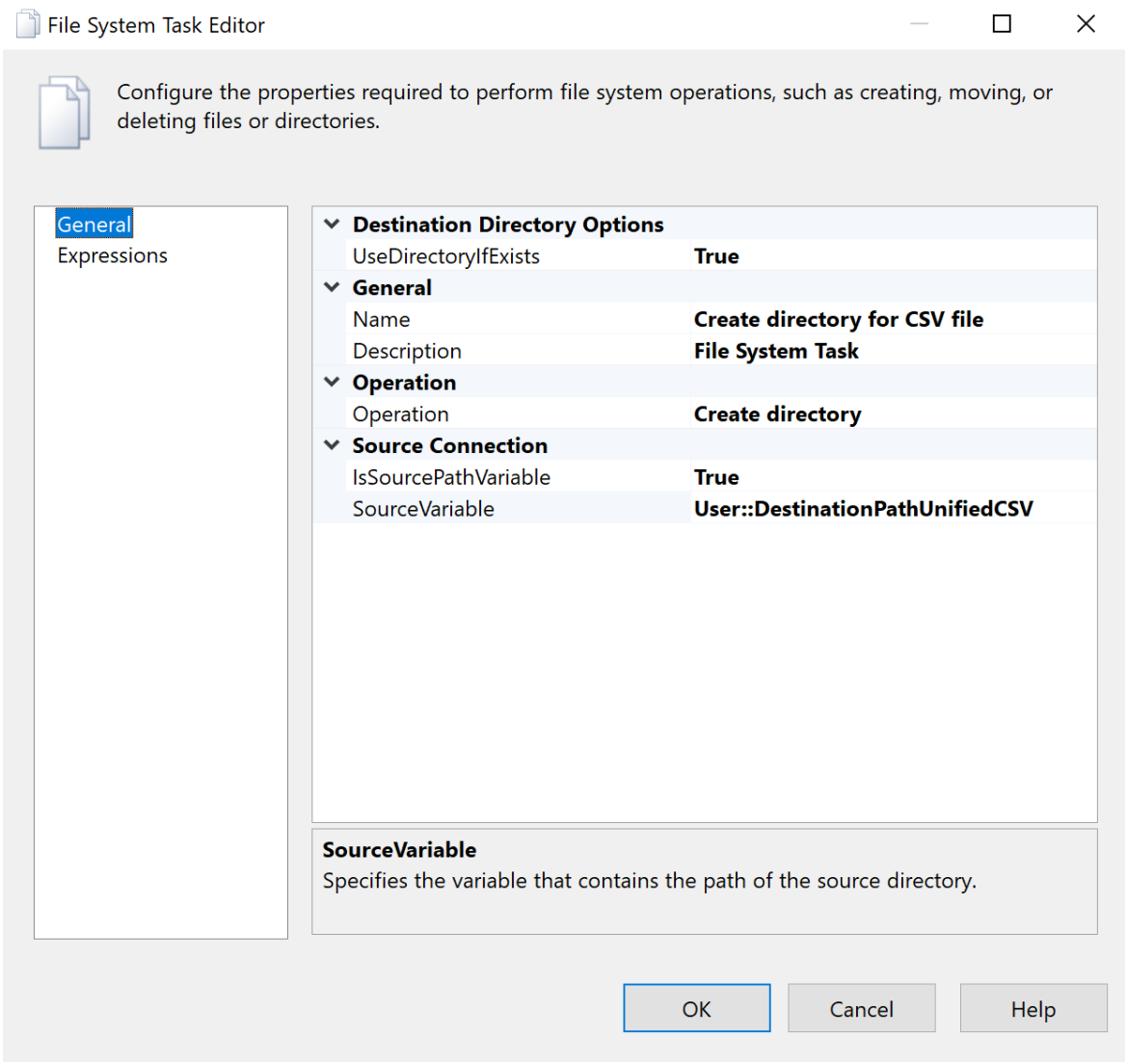


Figure 2: *File System task* for new directory creation

2.1.2 Deletion of directory content

As stated previously, SSIS packages may be run several times but data always must be consistent. Therefore, in the case of a previously created desired directory and loaded data inside it, we must ensure that all previous data is deleted and then new data loaded with the new execution. For that reason, a new *File System Task* named *Delete directory content* is created that uses the option *delete directory content* (hence the name of the task). The task uses a user-defined variable for the directory path mentioned in **section 2.1.1**. *File System Task Editor* is visible in **Figure 3**.

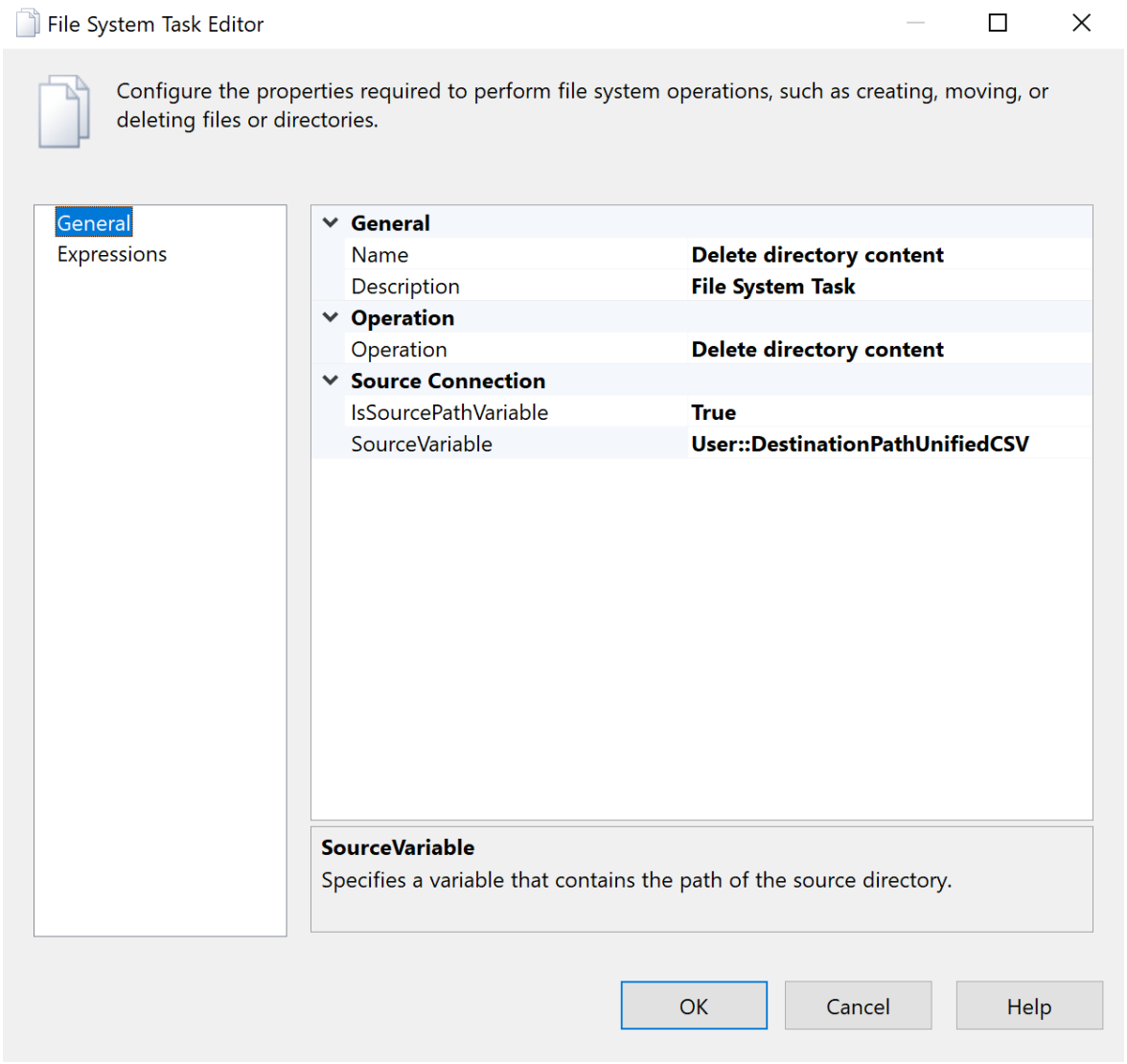


Figure 3: *File System task* for directory content deletion

2.1.3 Union of all CSV files

The last task of the *CSVFilePreparation* package is the *data flow task* named *Union 48 CSV files* that combines forty-eight CSV files using the *UNION ALL* transformation (see **Figure 4**). In *Connection Managers*, forty-eight connections to the source CSV files and one connection to the destination CSV file are created. The Connection to the destination file is converted to the project level so it can be used in the package named *DataImportingAndCleaning* (**subchapter 2.3**). Data flow components *Flat file source* (with proper source connection) and *Flat file destination* (with proper destination connection) are used to read and write flat data, respectively *(visible in* **Figure 4**). The destination CSV File is automatically created during the runtime of the aforementioned *CSVFilePreparation* package. Location is specified by a user-defined variable capturing directory path with the desired file name and proper extension (.csv). In the *Flat File*

Destination Editor, option to overwrite existing data is selected, meaning that each time the package runs, the file is emptied and new data is loaded. Certain columns were omitted during the selection of source columns in CSV files (the *green category contains extra columns that are not needed*).

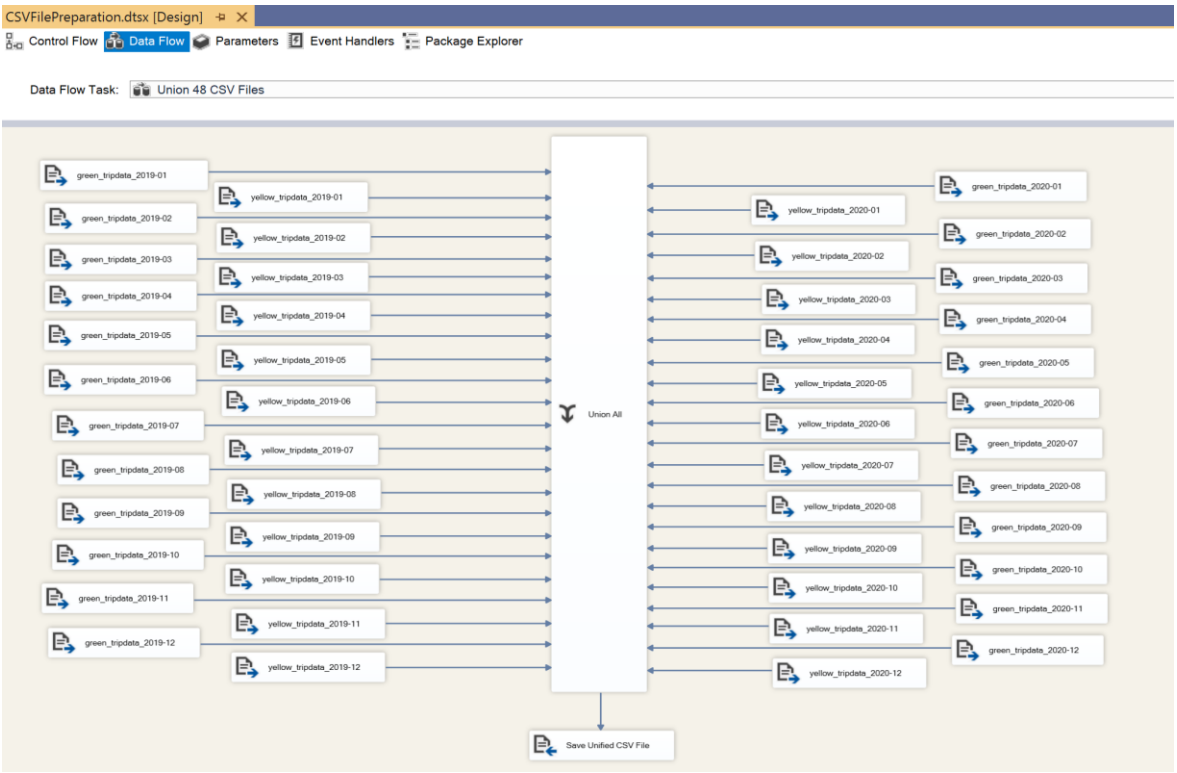


Figure 4: *Data Flow task* that combines 48 CSV files into one

2.2 Database and table preparation

By having *the unified CSV file* prepared in the next step it is proceeded to the data import to the SQL Server *(SQL Server is already introduced in **chapter 2**)*. Firstly, a decision needs to be made to know which columns from data source (*described in* **subchapter 2.1**) need to be imported to the table in the database. Then, the optimal data format regarding analytics and storage needs to be specified (*more on Power BI in **chapter 3***). Also, it is important to emphasize the need to have in mind the possibility of having calculated columns out of available data in the unified CSV file. Calculated columns serve to get pieces of the data field needed the most. When data is calculated out of certain fields, those fields may then also become redundant, as they may not be needed anymore at all. Before creating any tables and loading data into them, everything necessary should be prepared to properly partition tables. Partitioning requires addition of filegroups, and then a file must be added to each filegroup a file must be added. Furthermore, the partition function and the partition schema must be created before the creation of tables. Also, an additional table that

serves as a destination for rows not complying with constraints in initial data table is created (see **section 2.2.5**).

To perform these actions, the package *DatabaseTablePreparation.dtsx* is created. The Structure of the package is visible from **Figure 5**. All control flow tasks of the package are of the *Execute SQL Task* type. Tasks of this type require a connection to the server database and SQL code to be executed. SQL Scripts are directly input through *Task editor*. Specific tasks are grouped into *sequences* that define *control flow*. This *sequence control flow* is a subset of the package control flow.

All SQL code will be tested using *Microsoft SQL Server Management Studio 2019 (SSMS)* and the results of package executions. SSMS plays no other role except for testing.

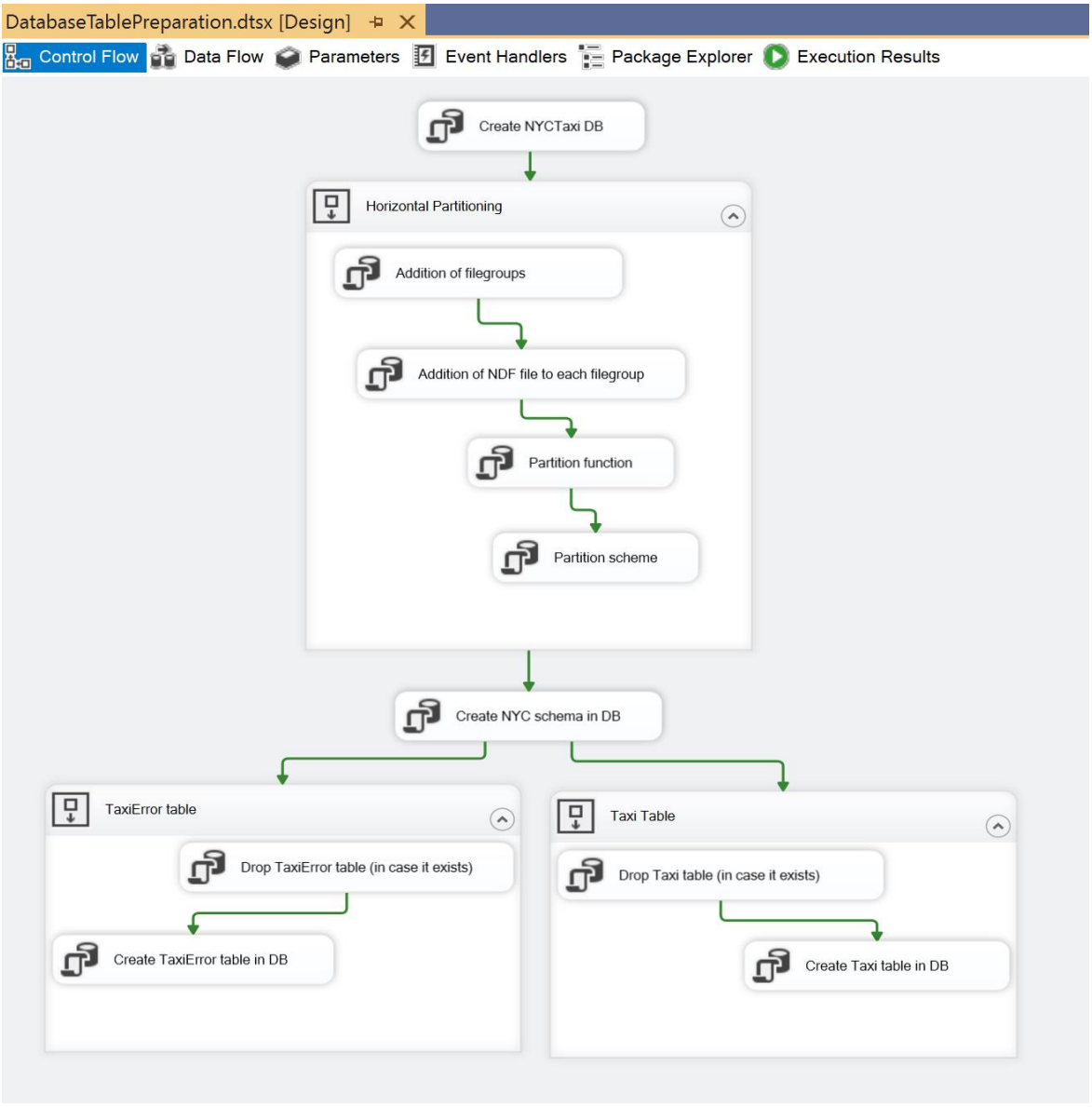


Figure 5: *DatabaseTablePreparation.dtsx* package structure

2.2.1 Creation of database in SQL Server

To be able to import data in a SQL Server, a SQL Server database must be initially created. The database can be created using the SSMS interface or by executing a T-SQL Script. T-SQL is an extension of SQL created by Microsoft to optimize executions and add extra options. T-SQL works only for the tools owned by Microsoft. In the Control Flow of the package, an *Execute SQL* task named *Create NYCTaxi is* created *DB.* Every SQL Server has its configuration database called a Master database. This database contains data about all other databases on the server. Therefore, in this specific task, an OLE DB connection to the master database will be made and the script shown in **Figure 6** will be run. The script checks if the database already exists. In the case it does not exist, it will create a new one. The script also specifies where a primary data file (with extension *.mdf*) and a transaction log file (with extension *.ldf*) are stored for the database (see **Figure 6**). Additional database configuration TSQL statements are executed but omitted in **Figure 6** for simplicity.

```
IF NOT EXISTS(SELECT * FROM sys.databases WHERE name = 'NYCTaxi')
  BEGIN

CREATE DATABASE [NYCTaxi]

 CONTAINMENT = NONE

 ON  PRIMARY
( NAME = N'NYCTaxi', FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\DATA\NYCTaxi.mdf' ,
  SIZE = 11GB , FILEGROWTH = 1GB )

 LOG ON
( NAME = N'NYCTaxi_log', FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\DATA\NYCTaxi_log.ldf' ,
  SIZE = 1GB , FILEGROWTH = 100MB )

  END
```

Figure 6: *Script* that *creates of NYCTaxi database*

2.2.2 Horizontal partitioning

Partitioning is one of the many advantageous methods of optimizing query performance when it comes to very large tables. Partitioning means that data in a table is divided into smaller parts that fit into a specific range. So, when it comes to retrieval of data, the table will not be scanned fully, but only a fraction of the data will be scanned. Consequently, queries take less time to execute. Horizontal partitioning means that rows are divided according to a specific range. It is opposite to vertical partitioning, where columns of one table are split into two or more tables. In the NYC taxi data, it is better to use horizontal partitioning because the data is arranged according to the datetime column specifying when it was collected. So, when it comes to filtering queries, the *DropOff_Date* column will be used most of the time in the WHERE statement. The WHERE statement is used to filter the data according certain conditions. There can be many conditions added by simply using logical operators (AND, OR, etc.). Partitioning is done before the table which is to be

partitioned is created and before any data is loaded. There are four important phases in successful partitioning that must be performed in the specific order. First, *File Groups* are created (it is possible to create just one, but that would make no sense in this specific case), then a *file* is added to each *File Group*. Next thing to do is to create of the *Partition function* with specified ranges. The Last step is to *Create a Partition Schema* with beforehand created *File Groups.* All steps described are visible in **Figure 5** under *Horizontal Partitioning Sequence*. Each partitioning step will be described in detail in upcoming subsections.

2.2.2.1 Addition of filegroups

A filegroup is in its essence a container (logical construct) that can be used to group database objects and database data files. The PRIMARY filegroup is the filegroup that contains the primary data file and any secondary files that are not put into other user-defined filegroups. All system tables are stored in the PRIMARY filegroup (for more info on filegroups see [11]). In the case a table is not partitioned, all data apart from the system tables goes into the PRIMARY filegroup as well. In terms of NYC Taxi. for each month in both 2019 and 2020 data a secondary filegroup (sometimes denoted as user-defined filegroup) is created. Before creating any filegroups, it is beneficial to check if there are filegroups already created under the same name. This is done because the script will be able to be executed several times without triggering any errors. In the case of the SSIS package that contains the *Execute SQL Task* running the SQL script used to add filegroups, checking if each filegroup exists prevents the whole package from failing. The ALTER DATABASE statement is used to change characteristic of a database, its underlying files and filegroups. The Script used to add one out of 24 filegroups is visible in **Figure 7**. The Other 23 scripts have the same format; only the name used to check if a filegroup exists and the name used to create a filegroup are different.

```
IF NOT EXISTS (SELECT * FROM sys.filegroups WHERE NAME='January_2019')
ALTER DATABASE NYCTaxi
ADD FILEGROUP January_2019
GO
```

Figure 7: One of the TSQL scripts adding filegroups to the database

2.2.2.2 Addition of secondary database file for each secondary filegroup

At this point, one database secondary file is added to each filegroup described in **subsection 2.2.2.1.** Database files are the actual objects in a computer system where the data is stored. They are a physical representation of filegroups. A filegroup can have one or many files. Files within filegroup can be distributed to many different memory disks.

There is always one default database file in the PRIMARY filegroup, the primary file. the primary file is the only file that has an MDF extension which stands for *Master Database File*. Every additional/secondary data file will have an NDF extension. It is important to know it because the extension is provided along with a file name in the string representing the directory path to the file created during the script execution visible in **Figure 8**. The same script is run only with modified naming for each month. In each filegroup only one secondary file is added. In total there will be 25 filegroups (including PRIMARY) and 25 data files (including MDF file). The Script consists of 24 SQL code blocks that check the existence of the file and add each to the proper filegroup is executed using *Execute SQL Task*. The task is visible in **Figure 5** in Sequence *Horizontal Partitioning*.

```
IF NOT EXISTS (SELECT   * FROM sys.master_files WHERE name = 'PartJanuary_2020')
ALTER DATABASE [NYCTaxi]
    ADD FILE
    (
    NAME = [PartJanuary_2020],
    FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\DATA\NYCTaxi_2020_01.ndf',
        SIZE = 500 MB,
        MAXSIZE = UNLIMITED,
        FILEGROWTH = 50MB
    ) TO FILEGROUP [January_2020]
GO
```

Figure 8: One of the TSQL Scripts adding a file to the filegroup

2.2.2.3 Partition Function

A Partition function defines how rows from the table are partitioned into logical divisions based on partition column values. Logical divisions are acquired by setting ranges using specific values. A Partition function sets the boundaries of each logical division. For that reason, the number of boundary values is always the number of partitions minus one. Those values are called the partition range. Every partition has one boundary that is included in the range and one that is not. The range can be right or left. If the range is right then an upper boundary is not included in the division and the first value in the division is the lower boundary (the smallest possible in that specific range). If the range is left then an upper boundary is included in the division and it is the last value of the division (the greatest possible in that specific range). Accordingly, a lower boundary is not included in the division itself. The Partition function is visible in **Figure 9**. The Partition function is executed by using Execute SQL Task in *Horizontal Partitioning* Sequence visible in **Figure 5**.

```
IF NOT EXISTS (SELECT * FROM sys.partition_functions WHERE name = 'PartitioningByYearAndMonth' )
CREATE PARTITION FUNCTION PartitioningByYearAndMonth (DATE)
    AS RANGE RIGHT FOR VALUES
        (   '2019-02-01',
            '2019-03-01',
            '2019-04-01',
            '2019-05-01',
            '2019-06-01',
            '2019-07-01',
            '2019-08-01',
            '2019-09-01',
            '2019-10-01',
            '2019-11-01',
            '2019-12-01',

            '2020-01-01',
            '2020-02-01',
            '2020-03-01',
            '2020-04-01',
            '2020-05-01',
            '2020-06-01',
            '2020-07-01',
            '2020-08-01',
            '2020-09-01',
            '2020-10-01',
            '2020-11-01',
            '2020-12-01'       );
```

Figure 9: Partition function setting ranges for logical divisions

2.2.2.4 Partition Scheme

The role of the partition scheme is to map a logical division to physical files contained within a filegroup. So, each partition (division) is mapped to each filegroup. It is possible to map all partitions to one filegroup. Filegroup consists of one or many files that can be spread on one or numerous memory disks, respectively (see **subsection 2.2.2.2**).
First, it is checked if a partition schema under the same name already exists, then, if it does not, a partition is created using SQL statement CREATE PARTITION SCHEME. Then the *partition function* described in **subsection 2.2.2.3** is used to properly distribute data rows into specific filegroups described in **subsection 2.2.2.1**. The Partition function is attached to the schema.
It is important to point out that the order of filegroups matters in the creation of the partition schema. SQL Server uses the order of filegroups previously set up in the partition scheme (see **Figure 10**) to properly assign the divisions created in the partition function (see **Figure 9**) to proper filegroups. The partition scheme is used when the table is created (see **section 2.2.5**) so when the data is loaded it is immediately stored in the proper partition. So, when the partitioning procedure is completed, it is then possible to create partitioned tables on the specific schema with a proper table column as the partition scheme argument (see **Figure 13** and **Figure 14**).

```
IF NOT EXISTS (SELECT * FROM sys.partition_schemes WHERE name = 'PartitionByYearAndMonth' )
CREATE PARTITION SCHEME PartitionByYearAndMonth AS PARTITION PartitioningByYearAndMonth
        TO (January_2019,
            February_2019,
            March_2019,
            April_2019,
            May_2019,
            June_2019,
            July_2019,
            August_2020,
            September_2019,
            October_2019,
            November_2019,
            December_2019,
            January_2020,
            February_2020,
            March_2020,
            April_2020,
            May_2020,
            June_2020,
            July_2020,
            August_2020,
            September_2020,
            October_2020,
            November_2020,
            December_2020  );
```

Figure 10: Partition scheme

2.2.3 Creation of SQL Server database schema

The next step was to create a database schema. A Schema is used as a logical collection of SQL Server database objects like tables, views, stored procedures, indexes, etc. (for more info on SQL schema see [8]) Firstly, it is checked whether a schema already exists, and then if not, the execution of the TSQL script (see **Figure 11**) is continued. The script is executed using *Execute SQL Task* and with the OLE DB connection to the database created in the previous task (see **section 2.2.1**). This notion of a schema is not to be confused with the one used in terms of database structure and tables within.

```
IF NOT EXISTS (SELECT * FROM sys.schemas WHERE name = 'NYC')
BEGIN
 EXEC('CREATE SCHEMA NYC')
 END
```

Figure 11: Creation of NYC Schema

2.2.4 Drop the table in case it exists

The next two *Execute SQL* tasks in their separate data flow sequences check whether tables described in **sections 2.2.5** and **2.2.6** exist and in case they do, it drops them. The

TRUNCATE statement could be used instead of the DROP statement, but it does not guarantee that the current table structure complies with the one needed. The DROP statement deletes the table data and the table itself, whereas TRUNCATE only deletes data but keeps the structure untouched. The script is visible in **Figure 12** and executed using the connection to the database created previously (see **section 2.2.1**). To drop the table that contains incorrect data, the same SQL statement shown in Figure 12 is used. Only the table name is substituted.

```
DROP TABLE IF EXISTS [NYC].[Taxi]
```

Figure 12: Drop table statement

2.2.5 Table creation

Finally, the last two *Execute SQL* tasks of the package deal with the creation of the table in which the data, previously stored in the unified CSV file, will be stored (see **section 2.1.3**) and the table in which stores redirected rows that triggered an error protocol (see **section 2.3.1**). As mentioned in **subchapter 2.1**, some columns from the unified CSV file are not created in the table because they are unwanted or they are used only to calculate values for new columns. The SQL Server database schema described in **section 2.2.3** is used along with the table name.

Columns *Rate_Code* and *Payment_Type* do not contain data of large format so they get less memory allocated (*tinyint* data type, visible in **Figure 13**). Columns like *Total_Amount*, *Trip_Distance*, *Fare_amount*, *Extra*, and calculated column *Trip_Duration* (described in **section 2.3.1** and **subsection 2.3.1.1**) get more memory allocated due to data size they may contain and also contain *precision 2* for the decimal data type (decimal and numeric are synonyms, precision means the number of decimal places). Column *Dropoff_Date* is a result of the *Dataflow* component described in **subsection 2.3.1.1**. The table allows NULL values for each column due to the possibility of corrupted or blank data in the source (will be explained in section **subsection 2.3.1.1**). The table is partitioned on the partition scheme described in **subsection 2.2.2.4** by using the *DropOff_Date* column.

```
CREATE TABLE [NYC].[Taxi](
    [Trip_Duration] [int] NULL,
    [Rate_Code] [tinyint] NULL,
    [Passenger_Count] [tinyint] NULL,
    [Trip_Distance] [float] NULL,
    [Fare_Amount] [float] NULL,
    [Extra] [float] NULL,
    [Tip_Amount] [float] NULL,
    [Total_Amount] [float] NULL,
    [Payment_Type] [tinyint] NULL,
    [DropOff_Date] [date] NULL
) ON [PartitionByYearAndMonth] ([DropOff_Date])
```

Figure 13: *Script used to create Taxi Table*

2.2.6 Table for rows that triggered the error protocol

OLE DB Destination Dataflow component loads data into SQL Server database. *OLE DB Destination* has built-in output to properly handle errors. It is usually a red precedence constraint that indicates this secondary path through which redirected rows flow into the secondary destination (in this case OLE DB destination table). For that reason, a new table is created as a destination for the rows that do not comply with data types, lengths or could not be used/transformed for derived columns in the original table. Also, two extra columns are added that SSIS provides in the error output. Those are *ErrorColumn* and *ErrorCode*, both numeric values. The rest of the column names are taken from the original table (see **section 2.2.5**) except that the data type of each column is set to be the string to resemble original data from the CSV file. The structure of the table for error rows is visible in **Figure 14**.

```
CREATE TABLE [NYC].[TaxiError](
    [Trip_Duration] [varchar] (50) NULL,
    [Rate_Code] [varchar] (50) NULL,
    [Passenger_Count] [varchar] (50) NULL,
    [Trip_Distance] [varchar] (50) NULL,
    [Fare_Amount] [varchar] (50) NULL,
    [Extra] [varchar] (50) NULL,
    [Tip_Amount] [varchar] (50) NULL,
    [Total_Amount] [varchar] (50) NULL,
    [Payment_Type] [varchar] (50) NULL,
    [DropOff_Date] [date] NULL,
    [ErrorCode] [int] NULL,
    [ErrorColumn] [int] NULL
) ON [PartitionByYearAndMonth] ([DropOff_Date])
GO
```

Figure 14: Script used to create TaxiError table

2.3 Data importing and cleaning

The next thing to do is to import the data from the CSV File (connection to its location already set as the project level connection in *Connections Manager*) to the database table described in **section 2.2.5**. Each package in the project must be able to be executed many times and each time to ensure data consistency. Therefore, the package that imports the data from the source CSV file to the destination database table must delete the content of the table in case the package is run independently from previous packages described in **subchapters 2.1** and **2.2**. This is accomplished by a simple TRUNCATE statement with a specified database table, so the description of this *Execute SQL task* will be skipped. Afterward, the data can be imported successfully. By having data imported it is possible to proceed to data cleaning which is a vital process of preparing data for analysis. Cleaning is not by definition deletion of corrupted data but more like a precautionary step, taken to increase data validity. The package *DataImportingAndCleaning.dtsx* is designed to import and clean data from the unified dataset. The structure of the package is visible in **Figure 15**. It is highly unrecommended to proceed to data analysis and visualization with unreliable data because it can lead to wrong conclusions and break the entire project. Thus, the data cleaning phase may be the most important step in the entire final project.

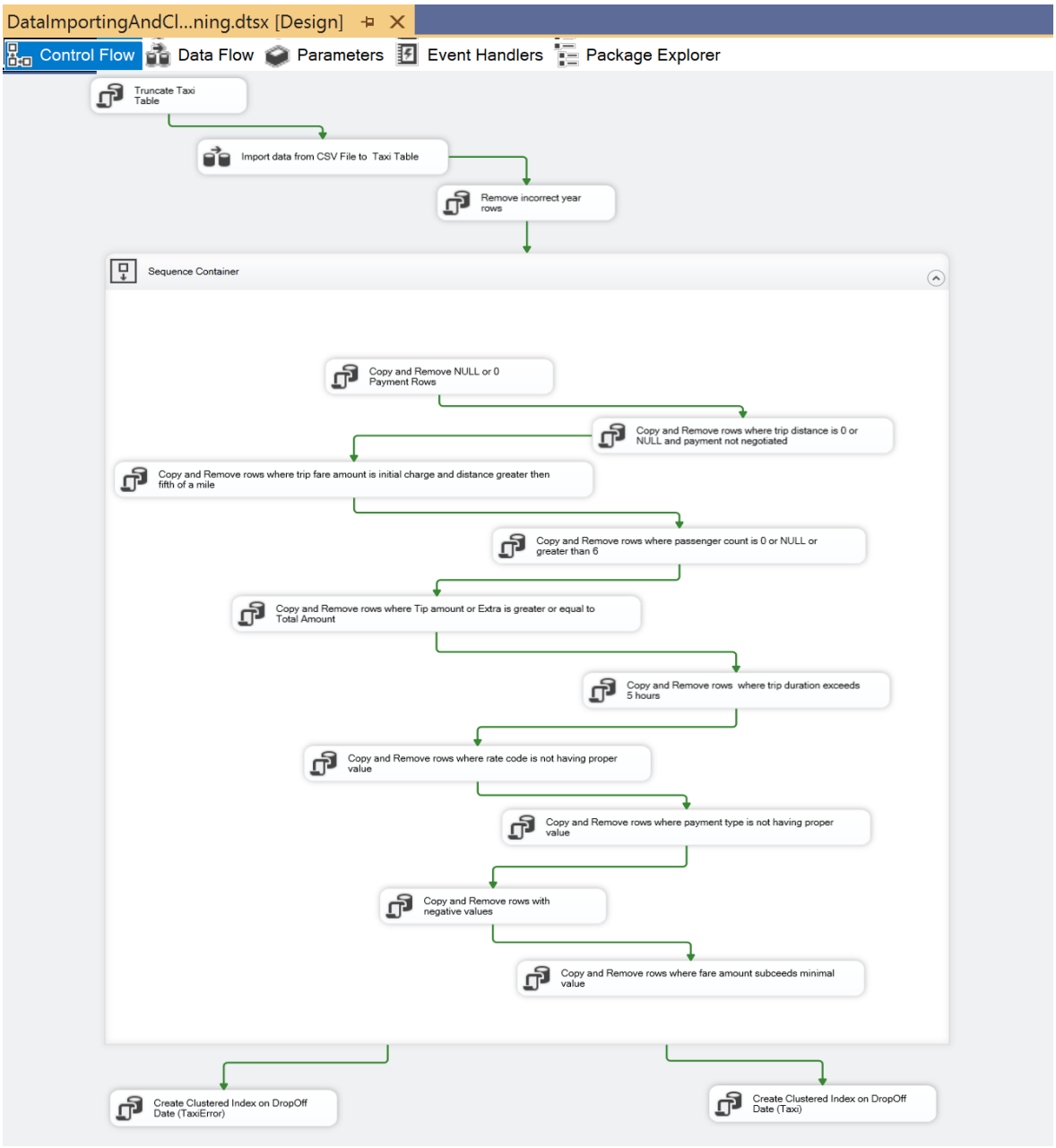


Figure 15: *Package importing and cleaning data, creating an index on tables*

2.3.1 Importing data from the CSV file to table

To accomplish importing data from flat file source to the SQL Server, the file prepared and the database table are prepared (see **chapters 2.1** and **2.2**). In the *DataImportingAndCleaning* package, a *Data Flow Task* is created that besides selecting only desired columns and importing data, plays important role in data conversion and calculation. It is what enabled to calculate new columns easily before unnecessary data is imported and used in calculations or extractions. The structure of the data flow is visible in **Figure 16**.

Two columns in the source file that captured the date and time when a taxi ride started and ended were used to calculate trip duration. The column that captures the date and time when the ride ended was also used to get the date only. That new piece of data will be used for table partitioning (partitioning described in **section 2.2.2)**.

To be able to perform calculations and import fields correctly columns *PickUp_Datetime* and *DropOff_Datetime* need to be converted to *DateTime* data types originally loaded as *String* data types. Then, in the *Derived column transformation* columns are calculated and derived. Here every blank field is replaced with a NULL value. In case if this step was not included then the columns from the dataset containing blank values would not be qualified to be converted to the wanted data type. In the last component, the data is finally imported to the database table using the same OLE DB connection used throughout the package *DatabaseTablePreparation*. The connection itself was initially described in **section 2.2.3**. In upcoming **subsections 2.3.1.1**, **2.3.1.2 and 2.3.1.3** three important components of the package will be described (see **Figure 16**). They deal with deriving columns and import errors reduction. Also, they give insights into errors that have occurred. Possible errors are incorrect values in the original CSV file that may be impossible to be converted into specific desired data types.

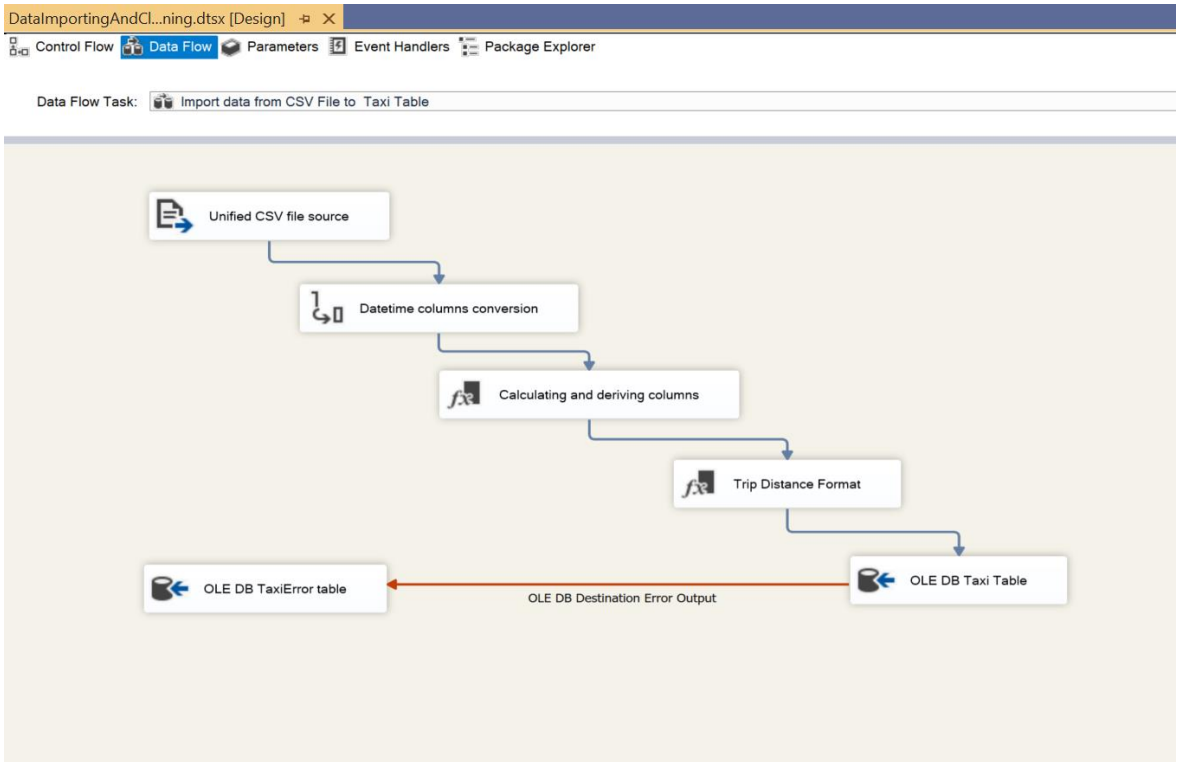


Figure 16: *Data flow task* loading, converting, calculating, and importing data

2.3.1.1 Calculating and deriving columns

In this *Dataflow* task (see **Figure 16**) the *Derived Column* component is used. It takes values from available columns, transforms them using a specified expression, and creates a new column. In the *Data conversion* component, previously columns *Dropoff_datetime* and *Pickup_datetime* were converted from *String* (every field in the CSV file is recognized as *String* by default, can be changed in the *flat file connection* itself but many errors can happen afterward) to *Datetime* datatype. Hence the name of the component. Converted *DateTime* columns will be used to calculate column *Trip_duration*. To accomplish this, the function DATEDIFF is used (stands for *date difference*). It is a basic SQL server function used to calculate how much time passed between two *DateTime* values. It also requires an interval argument. To get the result in seconds, "s" is passed as the interval argument. Also, to make sure not to get errors when a blank field is present in the source file, an expression is created that checks if the field is empty and if it is it will be replaced with the NULL value. For example, the case of blank string conversion to the numeric datatype would trigger the error protocol because the SQL server does not allow blank strings for a wide variety of datatypes. In the case it was a *String*, no error would be generated. For the complete structure of the transformation see **Figure 17**. The red precedence constraint (red arrow in **Figure 16**) indicates the flow of rejected rows and additional columns to the new destination table.

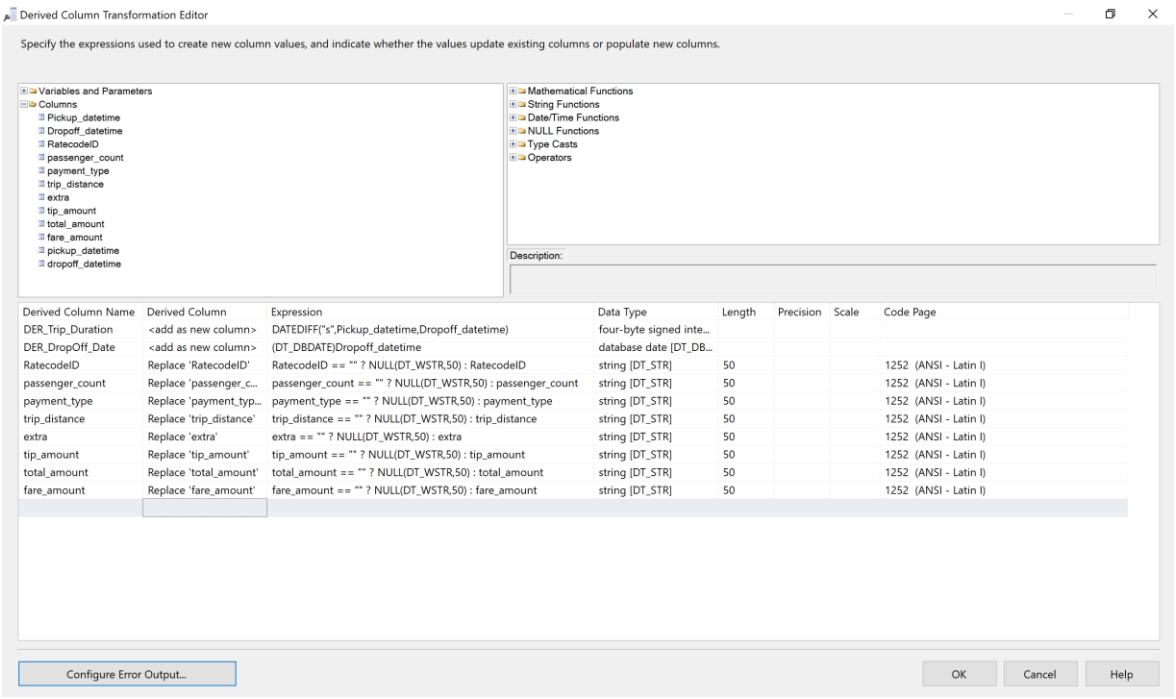


Figure 17: *Derived Column Transformation Editor* showing expressions used to derive and calculate columns

2.3.1.2 Trip Distance Format

In the unified CSV file, the *Trip_Distance* column was formatted in a way that in the case of its value being smaller than one only number from the right side of the decimal point were present. Zero was omitted. Therefore, another *Derived Column* transformation is created which concatenates zero to the left side of the decimal point in case the decimal point is the first character of the value. At this point, all of the columns are of the String data type except those representing dates (see **Figure 17**). The *Derived Column* transformation is not used to convert all columns to appropriate data types because the *OLE DB Destination* component converts it automatically while importing it to the SQL Server database table. The expression used to transform the *Trip_Distance* column is visible in **Figure 18**.

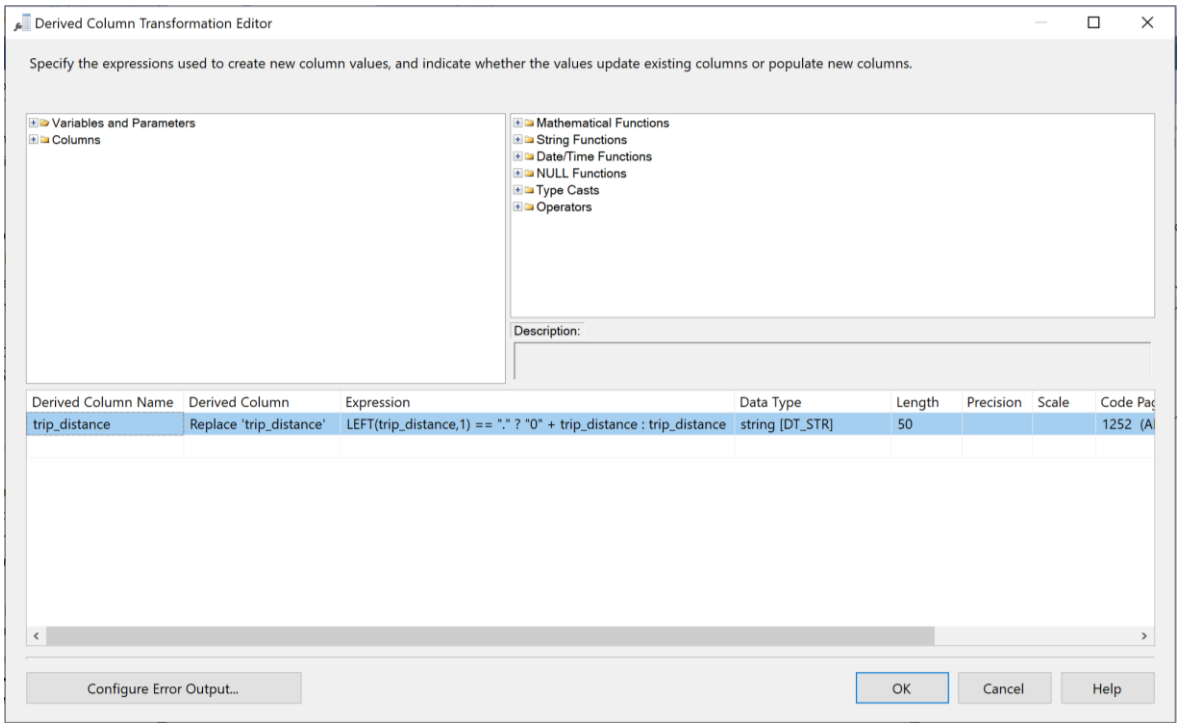


Figure 18: *Derived Column Transformation Editor* showing expression used to add 0 where needed

2.3.1.3 Saving rows triggering the error protocol

In the *OLE DB Destination component* (see **Figure 16**) that is the last step of data importing to the database table, the error output is redirected to another destination. A new table is created in the database (see **section 2.2.6**) to store rows that caused the error protocol to be triggered. It has the same structure as the original table, except that all columns from the original table are now of the *String datatype* in this new table. The reason behind it is that in the source file they were stored as plain text. Two extra columns

were added, *ErrorColumn* and *ErrorCode*, that store error code number and error column number provided by SSIS for each redirected row. *ErrorColumn* stores number unique to each execution of dataflow. The *ErrorCode* stores the number of the error that caused the row to be rejected in the first place. Structure of the table is visible from **Figure 14**. By analyzing this table after execution, it was possible to create or modify existing transformations to correct data to the extent it was possible (see **subsection 2.3.1.1**)

2.3.2 Cleaning the data using *Execute SQL Tasks*

To proceed with the data cleaning, the data must be understood. First, it must be assured that records in the table are collected in 2019 or 2020. That is done through checking the derived column *Dropoff_date* (derived in the dataflow task in **section 2.3.1**). Column *Total_Amount*, representing the amount a customer paid for the ride, must not be 0 or NULL. Records should not have negative values because they do not make sense for any column. Every ride must have at least one passenger and the maximum is six. Column *Extra* represents an additional charge so it can be 0 and every NULL value can be converted to 0. The Column *Total_Amount* must be always greater than all other charges, so all rows not complying with this condition must be removed. Columns checked for this condition are *Tip_Amount* and *Extra*. Trips that have a duration greater than 5 hours are very extreme and unlikely, so such rows can be removed. In the data dictionary it is obvious that for columns *Rate_Code* and *Payment_Type* there are predefined values, so any row not having a value from this set for the abovementioned columns should be removed, too. The final cleaning task is the task which removes all rows where the fare amount is less than the minimal possible amount in relation to the trip duration. All cleaning tasks are visible from **Figure 15**. Most of them are based on the data dictionary published by TLC (dictionary is published on Taxi & Limousine Commission official website, see [1]). Also, before removing any data from the dataset, the data selected for removing is saved into a separate table so that analysis of incorrect data can be performed. Every cleaning *Execute SQL task* will have the statement that saves incorrect rows to the table described in **section 2.2.6** except for the one removing incorrect years in **subsection 2.3.2.1**. Therefore, it will not be described separately in the upcoming sections.

2.3.2.1 Removing incorrect years

To remove data records in the SQL Server database table where the data was recorded not in *2019* or *2020*, the DELETE statement is used. It deletes rows from a table under certain conditions (if the condition was omitted all records would be deleted). To look only for the year part of the date, the SQL Server YEAR function is used. It returns an integer. The value of the integer represents the year of the specified date (see **Figure 19**).

```
DELETE
  FROM [NYCTaxi].[NYC].[Taxi]
  WHERE  YEAR(DropOff_Date) NOT IN ('2019','2020')
```

Figure 19: Statement that deletes rows not recorded in 2019/2020

2.3.2.2 Removing rows where payment is NULL or 0

Rows that have the *Total_Amount* column equal to 0 or is NULL are removed because every Taxi Ride in New York City has an initial charge of 2.5 USD (for more info on taxi fares see [9]). So, at some point in the collection of data, an error occurred. It could be also that a driver wanted to reach a target number of rides, therefore those rows do not contribute to the dataset quality needed for further analysis. The statement used to save and then remove unwanted rows is visible in **Figure 20**.

```
INSERT INTO [NYC].[TaxiError]
  SELECT *, NULL, NULL FROM [NYCTaxi].[NYC].[Taxi]
  WHERE  [Total_Amount] IS NULL OR [Total_Amount] = 0
GO
DELETE
  FROM [NYCTaxi].[NYC].[Taxi]
  WHERE  [Total_Amount] IS NULL OR [Total_Amount] = 0
GO
```

Figure 20: Statement that saves and deletes rows where payment is null or 0

2.3.2.3 Removing rows where trip distance is 0 or NULL

If the trip distance is 0 it can only mean that this trip in the real-life never happened, even though it was collected. The customer may have canceled the trip or the system collecting the data mistakenly recorded the ride. It can also mean that the driver wanted to reach the target number of rides, as was the case also in **subsection 2.3.2.2.** The statement used to save and remove unwanted rows is visible in **Figure 21**.

```
INSERT INTO [NYC].[TaxiError]
  SELECT *, NULL, NULL FROM [NYC].[Taxi]
  WHERE ([Trip_Distance] = 0 OR [Trip_Distance] IS NULL) AND  Payment_Type!=5
GO
DELETE
  FROM [NYC].[Taxi]
  WHERE ([Trip_Distance] = 0 OR [Trip_Distance] IS NULL) AND  Payment_Type!=5
GO
```

Figure 21: Statement that saves and deletes rows where trip distance is 0 or NULL

2.3.2.4 Removing rows where the trip fare amount is equal to initial charge and the distance greater than a fifth of a mile

The Fare_Amount column represents the fare amount of each trip. Fare is a price paid for a trip excluding additional costs and taxes. It is purely calculated based on trip length and duration (waiting time included) by a taximeter. So, any trip that has a trip distance greater than a fifth of a mile (same as 0.2 miles) has been charged at least 0.5 USD. For every 0.2 miles traveled additional 0.5 USD has been added. So, by adding up the initial charge of 2.5 USD and *Trip_Distance* greater than 0.2 miles it can be concluded that the trip must have a fare amount of at least 3.0 USD. For that reason, all rows that do not comply with this rule are removed. The statement used to save and remove rows not complying with this condition is visible in **Figure 22**.

```
INSERT INTO [NYC].[TaxiError]
  SELECT *, NULL, NULL FROM [NYC].[Taxi]
  WHERE  Fare_Amount=2.5 and Trip_Distance > 0.2
GO
DELETE
  FROM [NYC].[Taxi]
  WHERE  Fare_Amount=2.5 and Trip_Distance > 0.2
GO
```

Figure 22: Statement that saves and deletes rows where fare amount is equal to 2.5 USD and trip distance greater than 0.2 miles

2.3.2.5 Removing rows where passenger count is zero, NULL, or greater than six

The maximum number of allowed passengers in NYC taxis is five with one additional passenger in case it is a child under the age of seven (for more info on taxi trip rules see [10]). A taxi trip without any passengers does not make any sense and all such rows can be removed. So, rows with values 0, NULL, and greater than 6 can be safely deleted if they occupy any field for column *Passenger_Count.* The statement performing this action is visible from **Figure 23**.

```
INSERT INTO [NYC].[TaxiError]
  SELECT *, NULL, NULL FROM [NYCTaxi].[NYC].[Taxi]
  WHERE [Passenger_Count] = 0 OR [Passenger_Count] IS NULL OR [Passenger_Count]>6
GO
DELETE
  FROM [NYCTaxi].[NYC].[Taxi]
  WHERE [Passenger_Count] = 0 OR [Passenger_Count] IS NULL OR [Passenger_Count]>6
GO
```

Figure 23: Statement that saves and deletes rows where passenger count is 0, NULL, or greater than 6

2.3.2.6 Removing rows where tip amount or extra is greater or equal to the total amount

The total amount is the amount charged to the customer. Into this amount is calculated also the tip given by a customer who paid by credit card. The Tip amount is always 0 in case the customer paid with cash. The tip is a value also included in the total amount so those cannot ever be equal or greater. Extra is a sum of extra charges. The Extra is included in the total amount, therefore cannot be greater or equal to the total amount. The statement used to save and delete rows with the abovementioned conditions applied is visible in **Figure 24**.

```
INSERT INTO [NYC].[TaxiError]
  SELECT *, NULL, NULL FROM
  [NYC].[Taxi] WHERE [Tip_Amount]>=[Total_Amount] OR [Extra]>=[Total_Amount]
GO
DELETE
  FROM
  [NYC].[Taxi] WHERE [Tip_Amount]>=[Total_Amount] OR [Extra]>=[Total_Amount]
GO
```

Figure 24: Statement that deletes rows where tip amount or extra is greater or equal to the total amount

2.3.2.7 Removing rows where trip duration exceeds 5 hours

Trips exceeding 5 hours duration are extremely rare or not present at all in real life. By acknowledging that errors occur and sometimes *dropoff_datetime* values are sent after connection to the server is established, so it can be concluded that those values may be corrupted and falsely present duration time. The decision to go with 5 hours is taken due to the possibility of removing trips in the case a smaller value is chosen that had an extreme but realistic duration. Deletion is desirable regarding very extreme cases, whether they are or not correct. The statement used is visible in **Figure 25**. Five *hours* is equal to 18 000 *seconds*.

```
INSERT INTO [NYC].[TaxiError]
  SELECT *, NULL, NULL FROM [NYC].[Taxi]
  WHERE  Trip_Duration > 18000
GO
DELETE
  FROM [NYC].[Taxi]
  WHERE  Trip_Duration > 18000
GO
```

Figure 25: Statement that saves and deletes rows with a trip duration greater than 5 hours

2.3.2.8 Removing rows where rate code is not having proper value

From the data dictionary published on Taxi & Limousine Commission's official website (see [1]) it is clear that the *Rate_Code* column can have only six distinct values. Those are 1, 2, 3, 4, 5, and 6. Rows, where the rate code is one of the listed values, are kept. The Statement is visible in **Figure 26**.

```
INSERT INTO [NYC].[TaxiError]
  SELECT *, NULL, NULL FROM [NYC].[Taxi]
  WHERE Rate_Code NOT IN (1,2,3,4,5,6)
  GO
DELETE
  FROM [NYC].[Taxi]
  WHERE Rate_Code NOT IN (1,2,3,4,5,6)
  GO
```

Figure 26: Statement that saves and deletes columns with improper rate code

2.3.2.9 Removing rows where payment type does not have a proper value

As in **subsection 2.3.2.8**, from the data dictionary, it is obvious that the column *Payment_Type* also has only six distinct values. Those are 1, 2, 3, 4, 5, and 6. Rows, where this is not the case for the column *Payment_Type*, are removed. The statement is visible in **Figure 27**.

```
INSERT INTO [NYC].[TaxiError]
SELECT *, NULL, NULL FROM [NYC].[Taxi]
WHERE Payment_Type NOT IN (1,2,3,4,5,6)
GO
  DELETE
FROM [NYC].[Taxi]
WHERE Payment_Type NOT IN (1,2,3,4,5,6)
GO
```

Figure 27: Statement that saves and deletes columns with improper payment type

2.3.2.10 Removing rows with negative values

Negative values for any columns in the dataset do not make any sense because there are no real-life applications of them in the case of taxi data. The *Trip_duration* column cannot be negative because the smallest value is 0 (no duration at all). The *Rate_Code* column described in **subsection 2.3.2.8** has a defined set of values where none of them is negative. *Passenger_count* in case there are no passengers is 0, which is the smallest possible value (already discussed in **subsection 2.3.2.5**). *Trip_Distance* cannot be anything smaller than 0

(**subsection 2.3.2.4**). *Fare_Amount* is calculated by taximeter using time and distance, none of which is negative. *Tip_Amount*, *Extra*, and *Total_Amount* have the smallest value equal to 0, therefore cannot have negative values. The statement saving and deleting negative values is visible in **Figure 28**.

```
INSERT INTO [NYC].[TaxiError]
SELECT *, NULL, NULL FROM [NYCTaxi].[NYC].[Taxi]
  WHERE
        [Trip_Duration] < 0 OR [Rate_Code] < 0 OR
        [Passenger_Count] < 0 OR [Trip_Distance] < 0 OR
        [Fare_Amount] < 0 OR [Extra] < 0 OR
        [Tip_Amount] < 0 OR [Total_Amount] < 0 OR [Payment_Type] < 0
GO
DELETE
  FROM [NYCTaxi].[NYC].[Taxi]
  WHERE
        [Trip_Duration] < 0 OR [Rate_Code] < 0 OR
        [Passenger_Count] < 0 OR [Trip_Distance] < 0 OR
        [Fare_Amount] < 0 OR [Extra] < 0 OR
        [Tip_Amount] < 0 OR [Total_Amount] < 0 OR [Payment_Type] < 0
GO
```

Figure 28: Statement that saves and deletes rows with negative values

2.3.2.11 Removing rows where fare amount in relation to trip length is less than possible

Each trip has an initial charge of 2.5 USD, if the car is traveling faster than 12 miles per hour then one-fifth of a mile is charged 0.5 USD (1 mile is charged 2.5 USD). If the car is traveling slower than 12 miles per hour or is not in motion at all then every 60 seconds is charged 0.5 USD. So, the cheapest possible price in relation to trip duration is when the car traveled at a speed less than 12 miles per hour. See **Table 1** that shows calculated prices for certain speeds.

Table 1: Increase of speed increases the price if speed is greater than 12 miles per hour

Speed	Time	Distance	Price
0mph	60 sec	0 miles	0.5 USD
12mph	60 sec	0.2 miles	0.5 USD
24mph	60 sec	0.4 miles	1.0 USD
60 mph	60 sec	1 mile	2.5 USD

By reference to **Table 1,** it is noticeable that with the increase of speed over 12*mph* the price in relation to trip duration increases. So, the conclusion is that for any additional 60 seconds of the trip duration the smallest amount possible to be charged to the customer is

0.5 USD. In the dataset, any row that has a trip duration greater than or equal to 60 seconds and a fare amount less than 3.0 USD can be regarded as corrupted. So, for any row having a trip duration equal to some natural number *k* multiplied with 60 seconds, means that the fare amount is at least *k* multiplied by 0.5 USD plus the initial charge. Any row that does not comply with this result can be removed. Rows, where the fare is negotiated, are not taken into consideration. If the value of *Rate_Code* equals 5, then it is the negotiated fare meaning that the customer and the driver agreed-upon price of the trip in advance. They can do this only when the trip ends outside of the New York City, Westchester & Nassau Counties. Execute SQL Task is used in *the Control Flow* of the package visible in **Figure 15** to execute the *SQL statement* visible in **Figure 29**. This is the last *Execute SQL Task* used for the data cleaning.

```
INSERT INTO [NYC].[TaxiError]
SELECT *, NULL, NULL FROM [NYCTaxi].[NYC].[Taxi]
WHERE (( FLOOR([Trip_Duration] / 60) ) * 0.5) + 2.5 > [Fare_Amount] AND [Payment_Type] != 5
GO
DELETE
 FROM [NYC].[Taxi]
 WHERE (( FLOOR([Trip_Duration] / 60) ) * 0.5) + 2.5 > [Fare_Amount] AND [Payment_Type] != 5
 GO
```

Figure 29: Statement that saves and deletes rows subceeding minimal fare amount value

2.3.3 Creating Clustered Index

A clustered index is a type of index where data is physically reordered according to some column values in a table. It means that rows are physically stored close to one another according to the specified column value. In a table, there can be at most one clustered index. Data would be reorganized according to the values of specified columns in ascending or descending order. It is possible to choose one of the two possibilities. On the other hand, in the case of non-clustered index data is logically ordered (ascending or descending). The Logical order does not match how data is ordered physically on a disk. There may be many non-clustered indexes per single table for which additional index structure is created and stored in index pages. The structure itself is separate from data rows.

Usually, the clustered index helps with faster data retrieval when a column or columns used for indexing are also used in a WHERE clause, GROUP BY, or ORDER BY statement. The non-clustered index helps with values used in the GROUP BY statement, the WHERE clause, or the JOIN clause. As non-index takes additional space only the clustered index was considered. In the case of this final project paper a GROUP BY statement is used to summarize data by month and year. Having the clustered index did improve performance when using the GROUP BY statement compared to having a heap

(table without a clustered index). By comparing and testing many different Taxi table queries (generating results similar to those used in Power BI, see **chapter 3**) and their performances executed on the Taxi table as the heap table and with the clustered index, a significant improvement was observed with the creation of the clustered index.

2.3.3.1 Database and table context

The NYCTaxi database consists of tables *Taxi* and *TaxiError*. The *Taxi* table contains one hundred and ten million five hundred nine thousand thirty-two rows which are approximately 7.42 GB of data (after data is cleaned, see **section 2.3.2**). *TaxiError* Table contains six million three hundred twenty-one thousand six hundred ten rows which are approximately 0.38 GB of data. That is 4.68% of the initial data loaded. The TSQL statement used to create clustered index is visible in **Figure 30**.

```
CREATE CLUSTERED INDEX [ClusteredIndexDropOff_Date] ON [NYC].[Taxi]
( [DropOff_Date] ASC )

CREATE CLUSTERED INDEX [ClusteredIndexDropOff_Date] ON [NYC].[TaxiError]
( [DropOff_Date] ASC )
```

Figure 30: Two TSQL statements used to create clustered index

2.3.3.2 Query performance comparison

During the testing, the clustered index was created on the column *DropOff_Date*, also used for partitioning (see **section 2.2.2**). The clustered index does not require that column values are unique. It adds to every duplicate value of a clustering index an additional 4-byte integer value called an *uniqueifier (*for more info on indexes see [12]). To execute the query visible in **Figure 31** took 70% less time on average when having clustered index created on the *DropOff_Date* column. By comparing execution plans (available in SSMS) of two types of query executions (clustered index and heap) it is obvious that table/index scan is the key part affecting query performance. In a state of having no clustered indexes created, data is unordered (heap) and the entire table needs to be scanned to perform grouping.

```
SELECT COUNT(DropOff_Date)
FROM [NYCTaxi].[NYC].[Taxi]
GROUP BY MONTH(DropOff_Date),YEAR(DropOff_Date)
```

Figure 31: Performance test query

3 POWER BI

Power BI is a tool used for analytics and data visualization. In this final project paper Power BI will be used to properly visualize and then analyze and quantify how the pandemic of virus Covid-19 affected the use of Taxi Services in New York City. Due to the fact that the data has already been cleaned and formatted by using SSIS, Power BI will be only used to visualize data even though it is capable of cleaning and formatting data. It is also able to process large amounts of data (Big Data) as is the case in this paper, but only if data is properly structured.
For three hypotheses, there will be one report page that will include a line graph, a bar graph, and a table, each showing the specific value over the time starting from *01.01.2019* until *31.12.2020* grouped by month and year and for one hypothesis there will be a page consisting of a matrix table and a stacked chart. In the clustered column chart (bar graph) each month of 2020 will be compared to the same month in 2019 over a specific value. The table will show the rate of growth by comparing a month of 2019 to the same month in 2020 over a specified value. The table will also consist of the month name, the year 2019 specified value over the specified month, the year 2020 specified value over the same month, and the trend column which will signify whether growth is increasing or declining. The line graph will show how a specific value changed over time (during 12 months of 2019 and 12 months of 2020, over 24 months in total). Each month will be represented by a point in relation to the specified value amount in chronological order and will be consecutively connected by a straight line. Matrix table will be used to compare the monthly percentage of credit card versus cash payments for years 2019 and 2020. The stacked chart will be used to show the ratio of trip counts of the same months (*ex. January 2019 compared to January 2020*) in 2019 and 2020.

3.1 Preparation

Before proceeding to any visualizations and analyses, to properly load and visualize data certain steps must be performed. Power BI, when loading data from a SQL Server database will automatically recognize data types and set them up as they are in a database table. As for the NYC Taxi Trip table is already done and described in **section 2.2.5** no other transformations of the original data will be necessary for Power BI. Before creating any visualizations, data must be loaded. An additional *Date* table will be created so that it can be referenced by date column values from the datasets. It would not be possible to use the date column from the original dataset in any visualizations properly because Power BI requires unique and contiguous values in the date table in order to use DAX time intelligence functions. DAX, a language used over many Microsoft tools, is a set of different functions and operators which when combined can build very powerful

expressions which perform advanced calculations. Time intelligence functions are simply used to perform certain operations on data over some time period.

3.1.1 Data loading

By starting Power BI Desktop, to create any reports, source data must be loaded and that is done simply by selecting the option to get the data and properly selecting the source. To get data from the SQL Server, option Database is selected and then option SQL Server database is chosen. To connect to the database properly, credentials used to log in to the SQL Server must be also used in Power BI to load the data. The step to transform the data will be skipped and the option to load the data will be selected directly.

3.1.2 Creation of Date table

In the Data section of Power BI, there is an option named *New Table* which is a DAX Table Constructor. It opens up an input field for DAX expression. The Date table is created using the DAX expression consisting of table name and CALENDAR function which as its first input parameter requires a start date and as its second input parameter requires the end date. To dynamically select the minimal date from the dataset as the start date, the MIN function is used. To get the end date, similarly the MAX function is used, both passed as input parameters to the CALENDAR function. After the expression is run, a new table Date is created with only one column named *Date,* which will contain all the date values starting from the start date input parameter up until the end date input parameter. The DAX expression used to create the *Date* table is visible in **Figure 32**.

```
1  Date = CALENDAR(MIN('NYC Taxi'[DropOff_Date]),MAX('NYC Taxi'[DropOff_Date]))
```

Figure 32: DAX expression used to create a Date table

3.1.3 Setting up table relationships

To be able to use the newly created Date table, relationship from the original table to the Date table must be created. One element from the *Date* table may be connected to many elements in the NYC Taxi trip table, therefore it is a one-to-many relationship. Relationships are managed in the Model section of Power BI. To use data properly, all tables in a model should be connected properly to at least one table. So, a fact table should be connected to some dimension table or tables. The table that contains actual data recorded for taxi trips is the fact table, meaning that it contains data that will be analyzed, whereas dimension tables store dimensions that describe data in fact tables and will be

used in visualizations for proper grouping. In this final paper there will be two dimension tables (*Payment Type* and *Date*), one calculated table (*Payment Summarized*), and one fact table (NYC Taxi). The *Payment Type* table is created manually by using Power BI's *Enter Data* option. Created relations are visible in **Figure 33**.

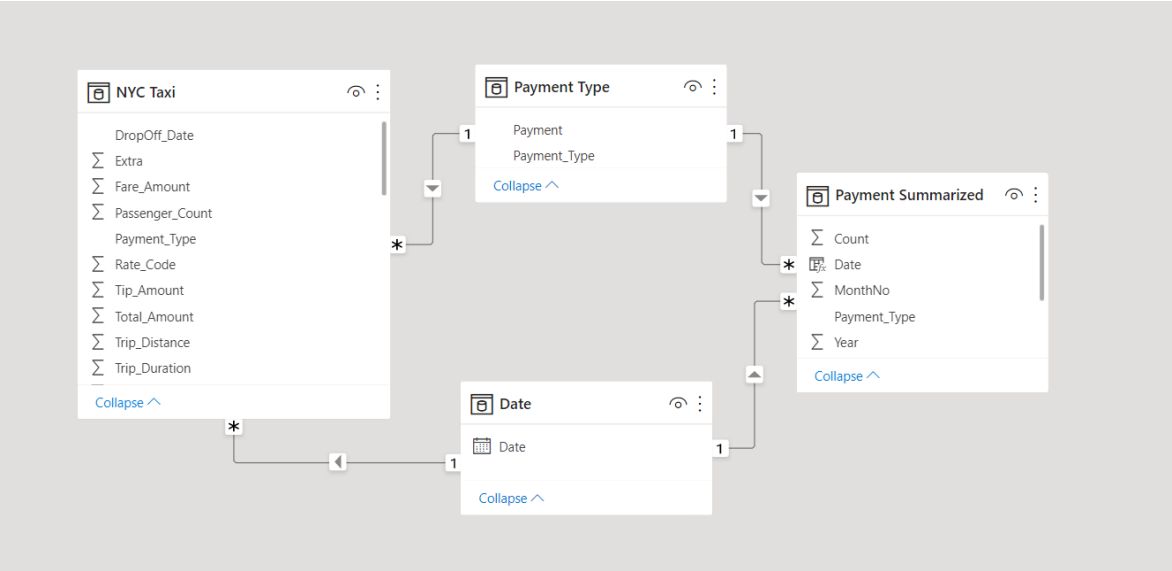


Figure 33: Relations of tables in the Model section

3.1.2 Calculated tables and measures

Power BI measures are calculated fields that are calculated during the runtime. So, no calculated data is stored as is the case with calculated columns. Calculated columns are DAX expressions processed before the data model in Power BI is created. In the case of measures, there is only a DAX expression for each measure which is evaluated every time the measure is used. Measures are useful when it comes to grouping and summarizing data. On the other hand, calculated columns are useful when new values are calculated for each row. It is possible to create new measures in the Power BI data or report section. Measures retrieve data in a specific and dynamic context, so it can be set by a slicer or specified when creating a visualization (by simply adding a specific hierarchy to the visualization). In the case of this project paper, it is a Year-Month or just Month hierarchy. A calculated table is created by selecting the *New Table* option in the Data section. It is defined by a DAX expression which performs operations over already loaded data. After it is created it should be connected to existing tables by setting up new relations. Final relations are visible in **Figure 33**. The DAX expression used to create the calculated table Payment Summarized is visible in **Figure 34**.

```
Payment Summarized =
    SUMMARIZE( FILTER('NYC Taxi', NOT ( 'NYC Taxi'[Payment_Type] IN {3,4,5 })), //table argument
        'Payment Type'[Payment_Type],'Date'[Date].[Year],'Date'[Date].[MonthNo], //columns
        "Count", COUNT('NYC Taxi'[Payment_Type])//calc. column
)
```

Figure 34: Creation of calculated table Payment Summarized

3.2 Visualizations and analysis

This chapter will contain visualizations and analysis of data which will be used to describe the results of hypotheses testing. A hypothesis can be confirmed or rejected. Hypotheses are defined in **subchapter 1.1**. To properly visualize data measures will be created. In this final project paper, measures will be used in the context of data grouped by Year and Month (by using the *Date* table) where in one measure there will be data for 2019 and in another the data for 2020. Differences between data collected during the pre-pandemic period (entire 2019, first two months of 2020) and the peri-pandemic period (last 10 months of 2020) will be quantified and visually presented.

3.2.1 Hypothesis 1

Hypothesis statement: There is significant difference in the amounts of taxi trips recorded during pre-pandemic months compared to peri-pandemic months.

3.2.1.1 Measures

To get the amount of taxi trips recorded in 2019 and 2020 by month the *Total Trips* measure is created to count the number of trips. The measure *Total Trips* is visible in **Figure 35**.

```
Total trips = COUNT('NYC Taxi'[Fare_Amount])
```

Figure 35: Total trips measure

To get the number of trips that were recorded in 2019 the measure *2019 Taxi Trip* is created. By using the DAX CALCULATE function which evaluates an expression in a modified context, it was possible to restrict the measure *Total Trips* (see **Figure 35**) to be evaluated only over data recorded in 2019. As the filtering function, the SAMEPERIODLASTYEAR is used which shifts one year back in a specific context (for

the Year 2019 context there is no data in this dataset). The measure *2019 Taxi Trip* is visible in **Figure 36**.

```
1  2019 Taxi Trips = CALCULATE([Total trips], SAMEPERIODLASTYEAR('Date'[Date]))
```

Figure 36: 2019 Taxi Trips Measure

To get the count of the data recorded in 2020, a new measure is created where measure *2019 Taxi Trips* is subtracted from the *Total Trips*. This specific measure works in context when the *Date* hierarchy in visual is restricted to Month or Year, but not both (see **Figure 37**).

```
1  2020 Taxi Trips = [Total trips]-[2019 Taxi Trips]
```

Figure 37: 2020 Taxi Trips Measure

To be able to express how much the values for months in 2020 are different from those in 2019 a new measure *Growth rate per Taxi Trips* is created. It simply calculates the percentage of difference based on a month in 2019 to a month in 2020. To get the result as percentage the measure is simply formatted to be Percentage in Power BI's Measure Tools. The measure is visible in **Figure 38**.

```
1  Growth rate Taxi Trips =
2      (('NYC Taxi'[2020 Taxi Trips]-'NYC Taxi'[2019 Taxi Trips])/'NYC Taxi'[2019 Taxi Trips])
```

Figure 38: Growth rate Taxi Trips measure

The Measure Trend Taxi Trips is created to specify if growth is negative or positive, or simply if more trips occurred in a 2019 month than in a 2020 month. It will be used for conditional formatting in the table visual. The measure is visible in **Figure 39**.

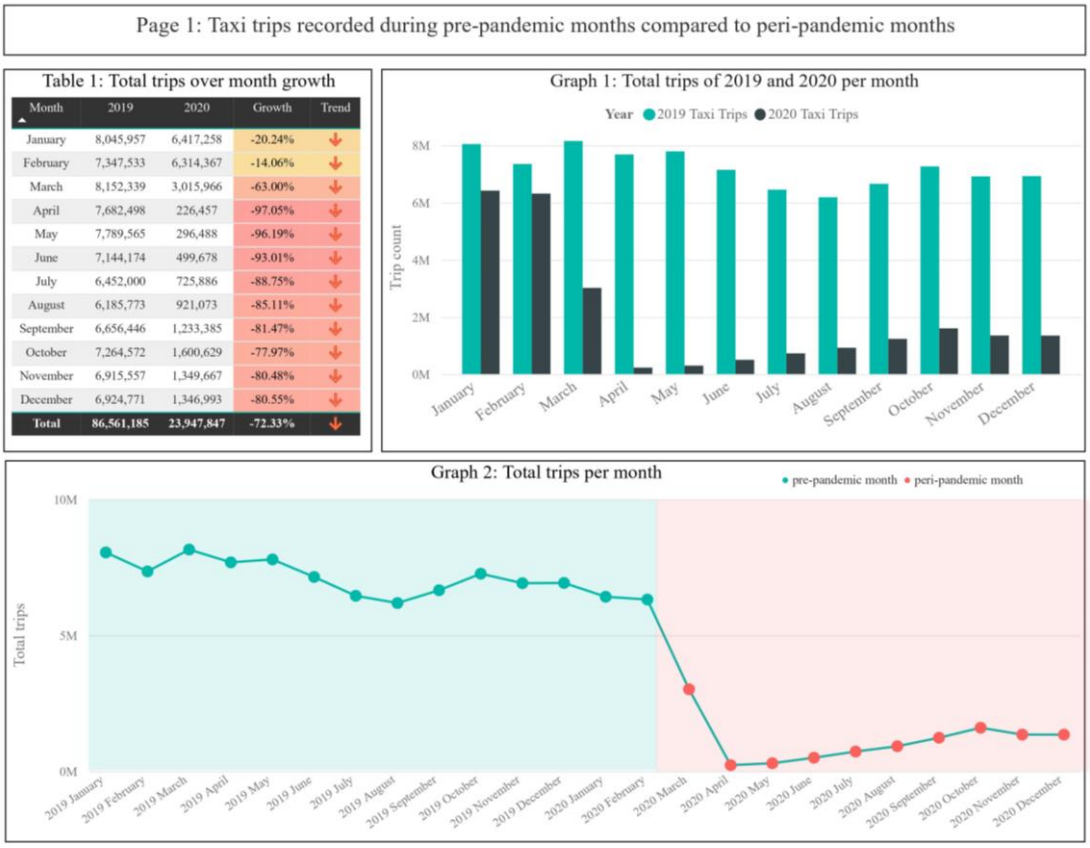
```
1  Trend Taxi Trips = IF ('NYC Taxi'[2019 Taxi Trips] >'NYC Taxi'[2020 Taxi Trips], 1, 0 )
```

Figure 39: Trend Taxi Trips measure

3.2.1.2 Visualizations

For Hypothesis 1 there is one report page (see **Figure 40**) which will include table, clustered column chart, and line chart visualizations. The table includes the columns Month, 2019, 2020, Growth, and Trend. It takes the *Month* hierarchy (*Date* table), *2019*

Taxi Trips measure, *2020 Taxi Trips measure*, *Growth rate Taxi trips* measure, and *Trend Taxi Trips* measure as its column values, respectively. Clustered column chart takes the *Month* hierarchy (*Date* table) as its horizontal axis and the measures *2019 Taxi Trips* and *2020 Taxi Trips* as its values. Therefore, its vertical axis shows the values of the trip count. *The* Line chart takes Date hierarchies Month and Year as its horizontal axis, therefore it shows values for the period from January 2019 to December 2020 chronologically. As its value, it takes the *Total trips* measure. The green shade in the line chart represents the pre-pandemic period and the red shade represents the peri-pandemic period. Report page 1 is visible in **Figure 40**.



Month	2019	2020	Growth	Trend
January	8,045,957	6,417,258	-20.24%	
February	7,347,533	6,314,367	-14.06%	
March	8,152,339	3,015,966	-63.00%	
April	7,682,498	226,457	-97.05%	
May	7,789,565	296,488	-96.19%	
June	7,144,174	499,678	-93.01%	
July	6,452,000	725,886	-88.75%	
August	6,185,773	921,073	-85.11%	
September	6,656,446	1,233,385	-81.47%	
October	7,264,572	1,600,629	-77.97%	
November	6,915,557	1,349,667	-80.48%	
December	6,924,771	1,346,993	-80.55%	
Total	86,561,185	23,947,847	-72.33%	

Figure 40: Report page 1

3.2.1.3 Results and conclusions

By observing visuals from Report page 1 visible in **Figure 40**, conclusions can be drawn on how pandemics of the Corona Virus (Covid-19) affected total counts of taxi trips recorded in pre-pandemic to peri-pandemic months. In Table 1 in **Figure 40**, Trip Count in January 2020 compared to January 2019 (both pre-pandemic) decreased by 20.24%. Similarly**,** Trip Count in February 2020 compared to February 2019 (both pre-pandemic) decreased by 14.06%. But, when March 2020 (peri-pandemic) is compared to March 2019

(pre-pandemic), decrease of 63.00% percent is observed. A state disaster emergency was declared on March 7, 2020, to fight COVID-19 in New York State (for more info on NY state disaster emergency see [13]). Also, on March 20, 2020, a State-wide stay-at-home order was declared (for more info on the aforementioned order see [14]). Similarly, Trip Count in April 2020 (peri-pandemic) compared to April 2019 (pre-pandemic) decreased by 97.05%. In Graph 1 and Graph 2 in **Figure 40**, it is evident that trip count after its lowest in April 2020 started to slowly increase until November 2020. In Graph 2, a significant dip happened in March 2020 and April 2020. March and April in 2020 are the first two months of the peri-pandemic period. The highest trip count in the peri-pandemic period was observed in October (see Graph 1), but if compared to the same period last year (October 2019), there is still a decrease of 77.97%.
Therefore, the conclusion can be drawn that there is a significant difference in the amounts of taxi trips recorded during pre-pandemic months compared to peri-pandemic months. The results supported the hypothesis 1.

3.2.2 Hypothesis 2

Hypothesis statement: There is significant difference in average time taken to arrive from pick-up to drop-off location during pre-pandemic months compared to peri-pandemic months in relation to trip length.

3.2.2.1 Measures

To get the difference in the average time taken to arrive from the pick-up to the drop-off location in relation to trip length, the average speed of every single trip is calculated. It is done by dividing trip distance by its duration (time taken). The duration column is explained in **subsection 2.3.1.1**. To get the total duration of each month in 2019 and 2020, the measure *Total Duration* is created. The measure is visible in **Figure 41**.

```
1  Total Duration = SUM('NYC Taxi'[Trip_Duration])
```

Figure 41: Total Duration measure

To calculate the speed, a total distance measure is needed. Total distance for each month is calculated by summing up the distances of each trip in a month (The Measure works for other *Date* hierarchies, but is used only in Month or Month-Year context). The *Total Distance* measure is visible in **Figure 42**.

```
1  Total Distance = SUM ('NYC Taxi'[Trip_Distance])
```

Figure 42: Total Distance measure

By having the measures *Total Distance* and *Total Duration* created, total distance and total duration measures for months in 2019 context and months in 2020 context can be created. To get the total duration for months in 2019, a new measure *2019 Total Durations* is created. The measure is visible in **Figure 43**.

```
1  2019 Total Durations = CALCULATE([Total Duration], SAMEPERIODLASTYEAR('Date'[Date]))
```

Figure 43: 2019 Total Durations measure

Similarly, measure *2019 Total Distances* is created. Measure visible in **Figure 44**.

```
1  2019 Total Distances = CALCULATE('NYC Taxi'[Total Distance], SAMEPERIODLASTYEAR('Date'[Date]))
```

Figure 44: 2019 Total Distances

By having measures *2019 Total Distances* and *2019 Total Durations* in place, the measure *2019 Speed* can be created. To get results in kilometers per hour (km/h), duration is converted from seconds to hours. Distance is already represented in kilometers. The measure *2019 Speed* is visible in **Figure 45**.

```
1  2019 Speed = 'NYC Taxi'[2019 Total Distances]/ ('NYC Taxi'[2019 Total Durations]/3600)
```

Figure 45: 2019 Speed measure

The *2020 Total Durations* measure is achieved by subtracting the *2019 Total Durations* measure from the *Total Durations* measure. The measure is visible in **Figure 46**.

```
1  2020 Total Durations = ([Total Duration])-[2019 Total Durations]
```

Figure 46: 2020 Total Durations measure

The *2020 Total Distance* measure is calculated by subtracting the *2019 Total Distance* measure from the *Total Distance* measure. The measure is visible in **Figure 47**.

```
1  2020 Total Distance = 'NYC Taxi'[Total Distance] - 'NYC Taxi'[2019 Total Distances]
```

Figure 47: 2020 Total Distance measure

Now, 2020 speed can be calculated using the *2020 Total Distance* and the *2020 Total Durations* measures. The measure is visible in **Figure 48**.

```
1  2020 Speed = 'NYC Taxi'[2020 Total Distance]/ ('NYC Taxi'[2020 Total Durations]/3600)
```

Figure 48: 2020 Speed measure

Growth rate Taxi Speed and *Trend Taxi speed* measures are created similarly as described in **subsection 3.2.1.1**, where instead of trips count speed measures are used. The *Speed* measure is created as a division of *Total Distance* by *Total Durations*.

3.2.2.2 Visualizations

For Hypothesis 2 there is one report page (see **Figure 49**) which will include table, clustered column chart, and line chart visualizations. The Table includes columns *Month*, *2019 Speed (km/h)*, *2020 Speed (km/h)*, *Growth* and *Trend*. It takes the Month hierarchy (the *Date* table), *2019 Speed measure*, *2020 Speed measure*, *Growth rate Taxi speed* measure, and *Trend Taxi speed* measure as its column values, respectively. Clustered column chart takes the Month hierarchy (the *Date* table) as its horizontal axis and the measures *2019 Speed* and *2020 Speed* as its values. Therefore, its vertical axis shows speed values. Line chart as its horizontal axis takes the Date hierarchies Month and Year, therefore it shows values for the period from January 2019 to December 2020 chronologically. As its value, it takes the *Speed* measure. The green shade in the line chart represents the pre-pandemic period and the red shade represents the peri-pandemic period. *Report page 2* is visible in **Figure 49**.

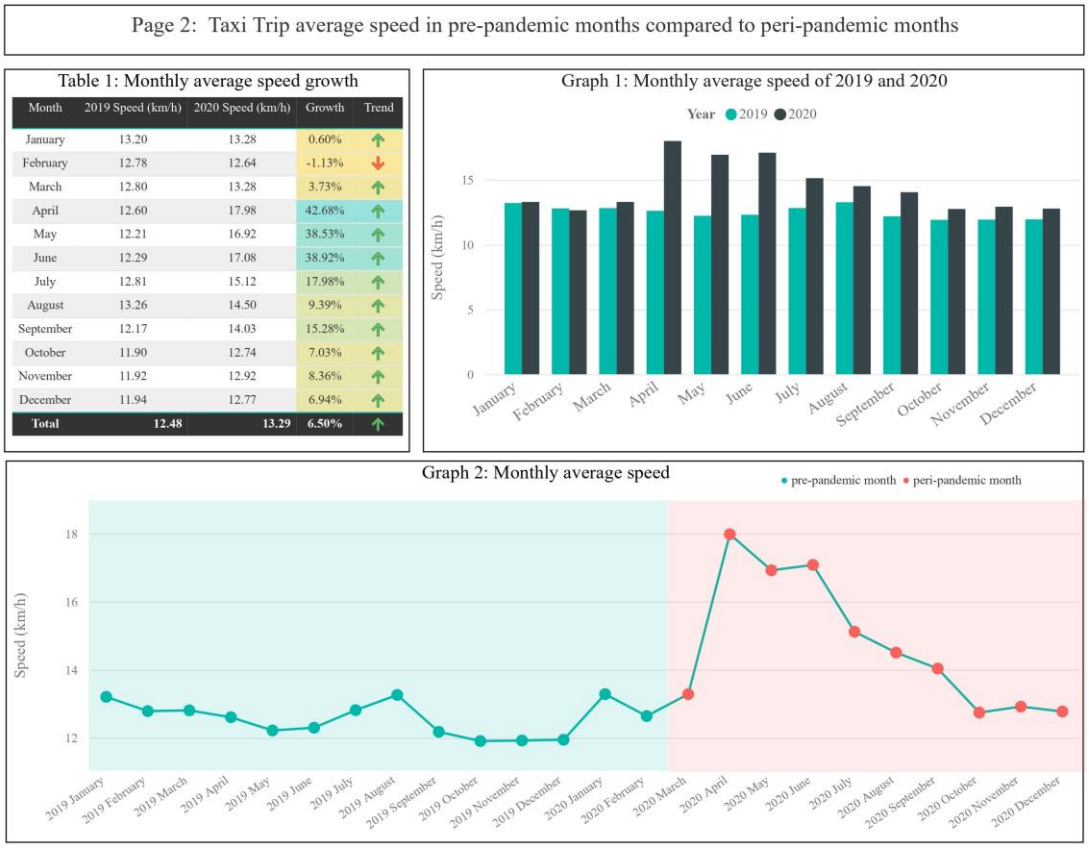


Figure 49: Report page 2

3.2.2.3 Results and conclusions

By observing visuals from Report Page 2 visible in **Figure 49**, conclusions can be drawn on how pandemics of Corona Virus (Covid-19) affected the average speed of taxi trips recorded in pre-pandemic to peri-pandemic months. In Table 1 in **Figure 49**, average speed in January 2020 compared to January 2019 (both pre-pandemic) increased by 0.60%. Also, the average speed in February 2020 compared to February 2019 (both pre-pandemic) decreased by 1.13%. But, when March 2020 (peri-pandemic) is compared to March 2019 (pre-pandemic), an increase of 3.73% percent is observed. A state disaster emergency was declared on March 7, 2020, to fight COVID-19 in New York State (for more info on NY state disaster emergency see [13]). The average speed in April 2020 (peri-pandemic) compared to April 2019 (pre-pandemic) increased by 42.68%. In Graph 1 and Graph 2 in **Figure 49**, it is evident that average speed after its highest in April 2020 started to decrease until November 2020. In December 2020 average speed decreased again compared to November 2020. In Graph 2, an average speed peak occurred in April 2020. Also, on March 30, 2020, a State-wide stay-at-home order was extended until April 29, 2020 (for more info see [15]). From Table 1 in **Figure 49** it is visible that all peri-pandemic months compared to the same month last year had an increase in average speed ranging from

3.73% to 42.68%. Speed increase occurred in relation to emptier roads during the peri-pandemic period (for more info see [16]).
Therefore, the conclusion can be drawn that there is a significant difference in the average time taken to arrive from pick-up to drop-off location during pre-pandemic months compared to peri-pandemic months in relation to trip length. The results supported the hypothesis 2.

3.2.3 Hypothesis 3

Hypothesis statement: There is significant difference in the amounts of cash payments compared to credit card payments during pre-pandemic and peri-pandemic months.

3.2.3.1 Measures

Measures for Report page 3 will reference the calculated table described in **section 3.1.2** and its expression visible in **Figure 32**. Even though the column *Count* exists in the calculated table *Payment Summarized* exists the column *Count*, to use it in measures it must be represented as a measure with the SUM function due to possible change of context. The measure *Sum Count* is visible in **Figure 50**.

```
1  Sum Count = SUM('Payment Summarized'[Count])
```

Figure 50: Sum Count measure

To get the same period last year in the Month context the measure *2019 Payment Count By Type* is created. The measure is visible in **Figure 51**.

```
1  2019 Payment Count By Type =
2          CALCULATE('Payment Summarized'[Sum Count], SAMEPERIODLASTYEAR('Date'[Date]))
```

Figure 51: 2019 Payment Count By Type

Now, the new measure *2020 Payment Count By Type* can be created by subtracting the measure 2019 Payment Count By Type from the measure Sum Count. The measure is visible in **Figure 52**.

```
2020 Payment Count By Type =
    'Payment Summarized'[Sum Count]- 'Payment Summarized'[2019 Payment Count By Type]
```

Figure 52: 2020 Payment Count By Type

3.2.3.2 Visualizations

For Hypothesis 3 there is one report page (see **Figure 53**) which will include matrix (pivoted table visualization) and 100% stacked column chart visualization. The Matrix (Table 1) includes columns CASH and CREDIT CARD with each having 2019 and 2020 sub-columns referencing the manually created table *Payment type* described in **section 3.1.3**. It takes the *Month* hierarchy (*Date* table) as rows. Its values come from measures *2019 Payment Count By Type* and *2020 Payment Count By Type*. Graph 1 (100% stacked column chart) in Page 3 as its horizontal axis takes the *Month* hierarchy (*Date* table). It takes measures *2019 Payment Count By Type* and *2020 Payment Count By Type* as its values.

Figure 53: Report page 3

3.2.3.3 Results and conclusions

By observing visuals from **Page 3** visible in **Figure 53**, conclusions can be drawn on how pandemics of Corona Virus (Covid-19) affected the ratio of cash payments and credit card payments recorded in pre-pandemic and in peri-pandemic months. In the Matrix visual (Table 1) for March 2020 (first month of the peri-pandemics period) out of the total taxi trips recorded, 25.85% were paid by cash, while in March 2019 27.12% were paid by cash. But then, in April 2020, the cash paid trips percentage grows to 39.99%. But at the same period last year (April 2019), the percentage of trips paid by cash was 28.20%. From Graph 1 it is observable that in April the ratio of trips recorded in 2020 has fallen drastically compared to the same period in 2019. In Table 1 in September 2019 (pre-pandemic) the percentage of trips paid by cash was 27.76%, which is smaller when compared to September 2020 (peri-pandemic) where the percentage of trips paid by cash was 28.18%. On the contrary, when October 2019 (pre-pandemic) percentage of trips paid by cash is compared to October 2020 (peri-pandemic), it is obvious that now percentage the value of the pre-pandemic month is greater than the percentage value of the peri-pandemic month. The percentage of trips paid by cash in October 2020 is 26.05%, whereas in October 2019 it is 27.25%. Except for the months April, May, June July, there is no significant difference in amounts of cash payments compared to credit card payments during pre-pandemic and peri-pandemic months.
The results did not support the hypothesis 3.

3.2.4 Hypothesis 4

Hypothesis statement: There is significant difference in the amounts of driver-reported passenger counts in a single trip during pre-pandemic months compared to peri-pandemic months.

3.2.4.1 Measures

To get a difference in the amount of driver-reported passenger counts in a single trip during the pre-pandemic months compared to the peri-pandemic months, a new measure that sums up passenger count is created. The *Passenger Sum* measure is visible in **Figure 54**.

```
1  Passenger Sum = SUM('NYC Taxi'[Passenger_Count])
```

Figure 54: Passenger Sum Measure

Now, a measure to get only data recorded in 2019 is created. The measure is visible in **Figure 55**.

```
1  2019 Passenger Sum = CALCULATE([Passenger Sum], SAMEPERIODLASTYEAR('Date'[Date]))
```

Figure 55: 2019 Passenger Sum measure

The *2020 Passenger Sum measure* is created by subtracting the *2019 Passenger Sum* measure from the *Passenger Sum* measure. It works for the Month hierarchy context (*Date* table). In the case the Month-Year hierarchy context was used then it would not be correct. The measure is visible in **Figure 56**.

```
1  2020 Passenger Sum = [Passenger Sum]- [2019 Passenger Sum]
```

Figure 56: 2020 Passenger Sum measure

To get the average number of passengers per single trip in a month of 2019, the new measure *2019 Passenger Average* is created. It is a result of the division of the *2019 Passenger Sum* measure by *2019 Taxi Trips* measure explained in **subsection 3.2.1.1**. The measure is visible in **Figure 57**.

```
1  2019 Passenger Average = 'NYC Taxi'[2019 Passenger Sum]/'NYC Taxi'[2019 Taxi Trips]
```

Figure 57: 2019 Passenger Average measure

Similarly, *2020 Passenger Average* measure is created. The measure is visible in **Figure 58**.

```
1  2020 Passenger Average = 'NYC Taxi'[2020 Passenger Sum]/'NYC Taxi'[2020 Taxi Trips]
```

Figure 58: 2020 Passenger Average measure

To be able to quantify by how much the average passenger counts for months in 2020 are different (increasing or decreasing) from those in 2019, the new measure *Growth rate Passenger* is created. It calculates the percentage of difference based on a month in 2019 to a month in 2020. To get the result as a percentage, the measure is simply formatted to be *Percentage* in Power BI's *Measure Tools*. The measure is visible in **Figure 59**.

```
Growth rate Passenger =
    (([2020 Passenger Average]-[2019 Passenger Average])/[2019 Passenger Average])
```

Figure 59: Growth rate Passenger measure

The Measure *Trend Passenger* is used to indicate if the growth rate is increasing or decreasing. It is created just for additional visualization purposes (similarly as in subsections 3.2.1.1 and 3.2.2.1). The measure is visible in **Figure 60**.

```
Trend Passenger = IF ('NYC Taxi'[2019 Passenger Average] >'NYC Taxi'[2020 Passenger Average], 1, 0 )
```

Figure 60: Trend passenger measure

3.2.4.2 Visualizations

For Hypothesis 4 there is one report page (see **Figure 61**) which will include table, clustered column chart and line chart visualizations. The table includes columns *Month*, *2019*, *2020*, *Growth*, and *Trend*. It takes the *Month* hierarchy (*Date* table), *2019 Passenger average* measure, *2020 Passenger average* measure, *Growth rate Passenger* measure, and *Trend Passenger* measure as its column values, respectively. Clustered column chart takes the *Month* hierarchy (*Date* table) as its horizontal axis and the measures *2019 Passenger average* and *2019 Passenger average* as its values. Therefore, its vertical axis shows values of the average passenger count. Line chart as its horizontal axis takes *Date* hierarchies Month and Year, therefore it shows values for period from January 2019 to December 2020 chronologically. As its value, it takes the *Passenger average* measure. The green shade in the line chart represents the pre-pandemic period and the red shade represents the peri-pandemic period. Report page 4 is visible in **Figure 61**.

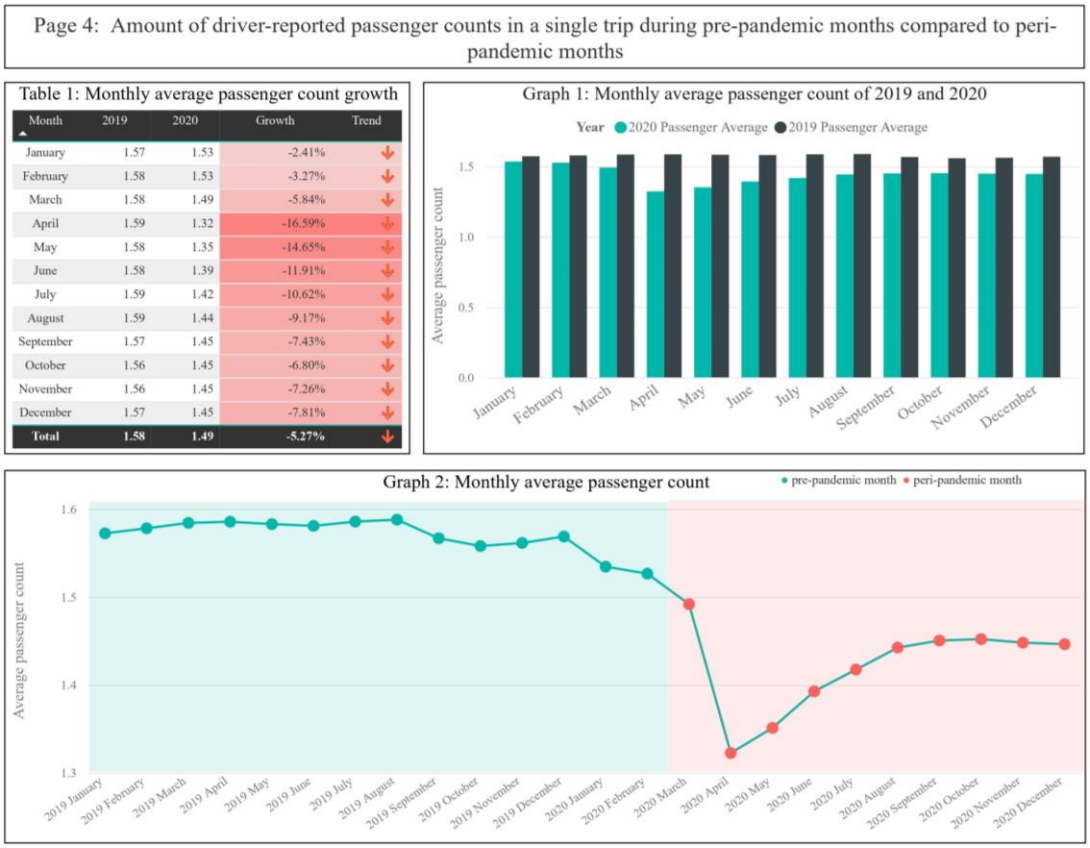


Figure 61: Report page 4

3.2.4.3 Results and conclusions

By observing visuals from Report page 4 visible in **Figure 61**, conclusions can be drawn on how pandemics of Corona Virus (Covid-19) affected average passenger counts of taxi trips recorded in pre-pandemic to peri-pandemic months. In Table 1 in **Figure 61** Average Passenger count in January 2020 compared to January 2019 (both pre-pandemic) decreased by 2.41%. Similarly**,** Trip Count in February 2020 compared to February 2019 (both pre-pandemic) decreased by 3.27%. But, when March 2020 (peri-pandemic) is compared to March 2019 (pre-pandemic), a decrease of 5.84% percent is observed. A state disaster emergency was declared on March 7, 2020, to fight COVID-19 in New York State (for more info on NY state disaster emergency see [13]). Also, on March 20, 2020, a State-wide stay-at-home order was declared (for more info on the aforementioned order see [14]). Then, the average passenger count in April 2020 (peri-pandemic) compared to April 2019 (pre-pandemic) decreased by 16.59%. In Graph 1 and Graph 2 in **Figure 61** it is evident that the average passenger count after its lowest in April 2020 started to slowly increase. In Graph 2, a significant dip happened during April 2020. April 2020 is the second month of the peri-pandemic period and the first month fully encompassing the state-wide stay-at-home order (for more info on the state-wide stay-at-home order see

[17]). The highest average passenger count in the peri-pandemic period was observed in March 2020 (see Graph 1), but if compared to the same period last year (March 2019) there is still a decrease of 5.84%. All peri-pandemic months had a smaller average passenger count per trip when compared to the same period last year.

Therefore, the conclusion can be drawn that there is significant difference in the amounts of driver-reported passenger counts in a single trip during pre-pandemic months compared to peri-pandemic months. The results supported the hypothesis 4.

4 CONCLUSION

In this final project paper, a large amount of NYC Taxi Trips data (Big data) was downloaded in 48 different CSV files. It was prepared, derived, imported to the SQL Server, and cleaned by using SQL Server Integration Services (SSIS). The next step was to properly load the data to Power BI, create additional dimensions and calculated tables, set up relations, and create measures. The last step was to properly quantify and visualize data so that results could be analyzed and hypotheses evaluated. Hypotheses were based on a comparison of pre-pandemic months compared to peri-pandemic months. All 2019 months were pre-pandemics months, including also January 2020 and February 2020. Therefore, the last 10 months of 2020 were regarded as peri-pandemic months as the state of emergency to fight COVID-19 pandemics was declared on March 7, 2020, in New York State. Hypotheses evaluation led to conclusions that there is a significant difference in the amounts of taxi trips recorded during pre-pandemic months compared to peri-pandemic months, there is a significant difference in the average time taken to arrive from pick-up to drop-off location during pre-pandemic months compared to peri-pandemic months in relation to trip length, there is no significant difference in amounts of cash payments compared to credit card payments during pre-pandemic and peri-pandemic months and that there is a significant difference in amounts of driver-reported passenger counts in a single trip during pre-pandemic months compared to peri-pandemic months. So, in the peri-pandemic period the number of trips recorded has fallen drastically, average driving speed has increased and the number of passengers per trip has fallen. Also, the results did not prove that there is a substantial change in the use of credit cards over cash as payment methods in the peri-pandemic period compared to the pre-pandemic period.

5 DALJŠI POVZETEK V SLOVENSKEM JEZIKU

V tej zaključni projektni nalogi je bila velika količina podatkov, po navadi imenovano po anglesko *Big data,* naložena v 48 različnih CSV datotekah (vrednosti, ločene z vejico). Podatki so bili pripravljeni, izpeljani, uvoženi v SQL strežnik in očiščeni z uporabo integracijskih storitev SQL strežnika (SSIS). Big Data kot izraz se običajno nanaša na opis velikih količin podatkov, ki jih ni mogoče niti uvoziti v orodja kot sta Microsoft Excel ali Google Sheets.

Naslednji korak je bil pravilno nalaganje podatkov v Power BI, ustvarjanje dodatnih tabel, nastavitev razmerij in ustvarjanje novih mer. Ker so bili podatki že očiščeni in formatirani, z uporabo SSIS, Power BI bo uporabljen samo za pravilno vizualizacijo. Vizualizacija bo uporabljena za analizo in količinsko opredelitev vpliva pandemije virusa Covid-19 na uporabo taksi storitev v New Yorku. Big Data (mnozični podatki) so uporabni za Power BI dokler so podatki strukturirani, kar pomeni da so opredeljeni v obliki tabel, sestavljenih iz vrstic in stolpcev z vzpostavljenimi relacijami in povezavami.

Zadnji korak je bil ustrezno kvantificirati in vizualizirati podatke, da bi lahko analizirali rezultate in ovrednotili hipoteze. Hipoteze so temeljile na primerjavi mesecev pred pandemijo v primerjavi s meseci pandemije. Vsi meseci leta 2019 so bili meseci pred pandemijo, vključno z januarjem 2020 in februarjem 2020. Zato so zadnjih 10 mesecev leta 2020 veljali za obdobja pandemije, saj je bilo izredno stanje za boj proti pandemiji COVID-19 razglašeno 7. marca 2020 v zvezni državi New York (za več informacij poglej [13]). Dne 20. marca 2020 je začela veljati tudi državna odredba o bivanju doma, ki omejujejo okoliščine v katerih lahko ljudje zapustijo svoje hiše (za več informacij glede prej omenjene državne odredbe poglej [14]).

Vrednotenje prve hipoteze je pripeljalo do zaključka, da obstaja velika razlika v številu taksi prevozov v mesecih pred pandemijo v primerjavi s meseci v obdobju pandemije. V tabeli 1 na sliki 39 število taksi prevozov se je januarja 2020 v primerjavi z januarjem 2019 (obe pred pandemijo) zmanjšalo za 20,24%. Podobno se je število prevozov februarja 2020 v primerjavi s februarjem 2019 (obe pred pandemijo) zmanjšalo za 14,06%. Ko pa primerjamo marec 2020 (obdobje pandemije) z marcem 2019 (obdobje pred pandemijo), je opaziti zmanjšanje za 63,00% odstotkov. Prav tako se je število taksi prevozov aprila 2020 (obdobje pandemije) v primerjavi z aprilom 2019 (obdobje pred pandemijo) zmanjšalo za velikih 97,05%. Največje število prevozov v obdobju pandemije je bilo opaziti oktobra, vendar je v primerjavi z istim obdobjem lani (oktober 2019) še vedno upad za 77,97%.

Vrednotenje druge hipoteze je pripeljalo do zaključka, da obstaja znatna razlika v povprečnem času, ki je potreben za prihod od mesta prevzema do mesta izstopa v mesecih pred pandemijo v primerjavi s meseci v obdobju pandemije glede na dolžino potovanja. V tabeli 1 na sliki 48 se je povprečna hitrost januarja 2020 v primerjavi z januarjem 2019 (oba pred pandemijo) povečala za 0,60%. Povprečna hitrost februarja 2020 v primerjavi s

februarjem 2019 (oba pred pandemijo) se je zmanjšala za 1,13%. Ko pa primerjamo marec 2020 (obdobje pandemije) z marcem 2019 (obdobje pred pandemijo), je opaziti povečanje za 3,73%. Povprečna hitrost aprila 2020 (obdobje pandemije) v primerjavi z aprilom 2019 (obdobje pred pandemijo) se je povečala za 42,68%. Iz tabele 1 na sliki 49 je razvidno, da se je v vseh mesecih v obdobju pandemije v primerjavi z istim mesecem lani povprečna hitrost povečala v razponu od 3,73% do 42,68%. Do povečanja hitrosti je prišlo v zvezi s bolj praznimi cestami v obdobju pandemije (za več informacij poglej [16]).

Vrednotenje tretje hipoteze je pripeljalo do zaključka, da v številu gotovinskih plačil v primerjavi s plačili s kreditnimi karticami v mesecih v obdobju pred pandemijo in v mesecih v obdobju pandemije ni bistvene razlike, razen aprila, maja, junija in julija. Aprila 2020 (obdobje pandemije) je odstotek gotovinskih plačanih prevozov 39,99%. Toda v istem obdobju lani (april 2019) je odstotek gotovinskih plačanih prevozov znašal 28,20%. Po aprilu je bila razlika med meseci v obdobju pandemije in pred pandemijo vedno manjša. Septembra 2019 (obdobje pred pandemijo) je odstotek gotovinskih plačanih prevozov znašal 27,76%, kar je manj v primerjavi s septembrom 2020 (obdobje pandemije), kjer je bil odstotek gotovinsko plačanih prevozov 28,18%. Odstotek gotovinskih plačanih prevozov oktobra 2019 (obdobje pred pandemijo) v primerjavi z oktobrom 2020 (obdobje pandemije) je očitno večji kar je nasprotno v primerjavi s septembrom 2019 in 2020. Odstotek prevozov plačanih z gotovino v oktobru 2020 je 26,05%, v oktobru 2019 pa 27,25%.

Vrednotenje četrte hipoteze je pripeljalo do zaključka, da obstaja velika razlika v povprečnem številu potnikov vdeleženih v enem prevozu v mesecih pred pandemijo v primerjavi s meseci v obdobju pandemije. Vsi meseci v obdobju pandemije so imeli manjše povprečno število potnikov na potovanje v primerjavi z enakim obdobjem lani. Najvišje povprečno število potnikov v obdobju pandemije je bilo zabeleženo marca 2020, vendar je v primerjavi z istim obdobjem lani še vedno upad za 5,84%.

5 REFERENCES

[1] Taxi & Limousine Commission. (n.d.). TLC Trip Record Data - TLC. Retrieved June 27, 2021, from https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page *(Quoted on pages 2, 3, 22 and 26)*

[2] West, Melanie Grayce (March 2, 2020). *"First Case of Coronavirus Confirmed in New York State"*. The Wall Street Journal. ISSN 0099-9660. *(Quoted on page 2)*

[3] Dzhanova, Yelena (April 10, 2020). *"New York state now has more coronavirus cases than any country outside the US"*. CNBC. *(Quoted on page 2)*

[4] Giuffo, J. (2013, October 1). *NYC's New Green Taxis: What You Should Know*. Forbes. https://www.forbes.com/sites/johngiuffo/2013/09/30/nycs-new-green-taxis-what-you-should-know/?sh=65a3e06832a2 (*Quoted on page 3)*

[5] Microsoft. (n.d.). *Power BI Desktop—Interactive Reports | Microsoft Power BI*. Power BI. Retrieved June 28, 2021, from https://powerbi.microsoft.com/en-us/desktop/ (*Quoted on page 3)*

[6] Peterson, R. (2021, October 7). *SSIS Tutorial for Beginners*: *What is, Architecture, Packages*. Guru99. https://www.guru99.com/ssis-tutorial.html (*Quoted on page 4)*

[7] AfterAcademy. (2020, May 1). *What is the difference between SQL and SQL server?* https://afteracademy.com/blog/what-is-the-difference-between-sql-and-sql-server (*Quoted on page 4)*

[8] Gupta, R. (2021, May 12). *A Walkthrough of SQL Schema. SQL Shack - Articles about Database Auditing, Server Performance, Data Recovery, and More.* https://www.sqlshack.com/a-walkthrough-of-sql-schema/ (*Quoted on page 14)*

[9] Taxi & Limousine Commission. (n.d.). *Taxi Fare - TLC*. Taxi & Limousine Commission (TLC). Retrieved July 11, 2021, from https://www1.nyc.gov/site/tlc/passengers/taxi-fare.page (*Quoted on page 23)*

[10] Taxi & Limousine Commission. (n.d.). *Passenger Frequently Asked Questions - TLC*. TLC. Retrieved July 12, 2021, from https://www1.nyc.gov/site/tlc/passengers/passenger-frequently-asked-questions.page (*Quoted on page 24)*

[11] Microsoft. (2021, May 25). *Database Files and Filegroups - SQL Server*. Microsoft Docs. https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-files-and-filegroups?view=sql-server-ver15 (*Quoted on page 11)*

[12] Ufford, M. (2021, August 16). *Effective Clustered Indexes*. Simple Talk. https://www.red-gate.com/simple-talk/databases/sql-server/learn/effective-clustered-indexes/ *(Quoted on page 29)*

[13] New York State. (2021, June 23). *Governor Cuomo Announces New York Ending COVID-19 State Disaster Emergency on June 24*. The Official Website Of New York State. https://www.governor.ny.gov/news/governor-cuomo-announces-new-york-ending-covid-19-state-disaster-emergency-june-24 *(Quoted on pages 35, 39, 45 and 48)*

[14] New York State. (2020, March 20). *Governor Cuomo Signs the "New York State on PAUSE" Executive Order*. The Official Website Of New York State. https://www.governor.ny.gov/news/governor-cuomo-signs-new-york-state-pause-executive-order *(Quoted on pages 36 ,45 and 48)*

[15] Raleigh, N.C. (2020, March 27). *Governor Cooper Announces Statewide Stay at Home Order Until April 29 | NCDHHS*. NCDHHS. https://www.ncdhhs.gov/news/press-releases/2020/03/27/governor-cooper-announces-statewide-stay-home-order-until-april-29 *(Quoted on page 39)*

[16] Berger, P., & Jones, C. (2020, December 19). *New York City Traffic Deaths Rise During Covid-19 Pandemic*. WSJ. https://www.wsj.com/articles/new-york-city-traffic-deaths-rise-during-covid-19-pandemic-11608382800 *(Quoted on page 40)*

[17] New York State. (2020b, June 23). *Governor Cuomo Announces New York Ending COVID-19 State Disaster Emergency on June 24*. The Official Website Of New York State. https://www.governor.ny.gov/news/governor-cuomo-announces-new-york-ending-covid-19-state-disaster-emergency-june-24 *(Quoted on page 46)*