Department of Management Science & Technology

MSc in Business Analytics

«Developing a platform for virtual data integration in big data environments»

By

RAPANOU Argyro & SOULIOU Georgia

Rapanou Argyro Student ID Number: BAPT1725

Souliou Georgia Student ID Number: BAPT1728

Name of Supervisor: CHATZIANTONIOU Damianos

March 2020

Athens, Greece

Abstract

We live in the era of Big Data, where the concern for data management is crucial for all organizations. This generates the need of sustainable infrastructures that will manipulate the data with speed and accuracy.
The current study attempts to investigate the virtual data integration in Big Data Environments, by describing the design, implementation and evaluation of a platform used to aid data exchange and preparation as well the definition of on demand data models that would assist on decision making.
Elements of the Platform are presented in a user friendly manner, that integrate multiple heterogeneous data inputs, creating a virtual homogeneity of information representation through the data canvas.
The Platform for Virtual Data Integration has been developed to integrate any kind of data input to a first consolidated layer that is presented in graph- based schema. The intent is the system to allow its users share and exchange information and work together a single operating entity and integrated planning unit.
The conceptual model of the schema is described, with the creation of a graph representation where attributes are mapped to entities. The key focus is the extent of Key-Value pairs to Key-Lists pairs.
All methods of development and their requirements are provided in detail along with a step-by-step installation of the tool.
Finally, the project is evaluated by its potential and all the future work that may be done on accessing the Bid Data world in more fast and consistent way.

Acknowledgments

We would like to express our sincere gratitude to Dr. Damianos Chatziantoniou, Associate Professor and Director of MSc in Business Analytics, Department of Management Science and Technology of Athens University of Economics and Business.

We are grateful for our collaboration in this Thesis Project as well as the cooperation for the last couple of years, for the duration of this Master's degree courses. We managed to conduct successfully this project with continuous support and effort.

Finally, we would like to express our sincere appreciation to our parents for encouraging and supporting us throughout our studies.

Table of Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

Abbreviation/ Acronym	Description
ER	Entity Relationship Model
PVDI	Platform for Virtual Data Integration
ML	Machine Learning
DB	Database
KL	Key-Lists
BI	Business Intelligence

1. Introduction

Background Information

In the current thesis report, we are interested in Data Integration and specifically in virtual Data Integration or more simply in Data Virtualization.
According to Wikipedia, Data integration [R1] involves combining data residing in different sources and providing users with a unified view of them. Data integration appears with increasing frequency as the volume (that is, big data) and the need to share existing data explodes.
With Virtual Data Integration, we give the illusion that data sources have been integrated without materializing data. That is the Data virtualization [R2] which according to Wikipedia is any approach to data management that allows an application to retrieve and manipulate data without requiring technical details about the data, such as how it is formatted at source, or where it is physically located, and can provide a single customer view (or single view of any other entity) of the overall data.
Data Virtualization [R3] technology is rapidly gaining momentum and is radically improving the productivity of users and developers to access distributed data sources for data integration, reporting and analytics, and application development. It is an agile method that enables real-time access to disparate data sources (on premise and in the cloud) in a fraction of the time and at a fraction of the cost of traditional approaches. Data virtualization extracts data from multiple disparate sources creating a unified virtual data layer that provides users easy access to the underlying source data.

Objectives

The objective is to describe the data infrastructure of an organization at a conceptual model, much like the traditional Entity-Relationship Model (ER). However, while ER uses a top-down approach where real-world entities and their relationships are depicted and utilized in a process that leads to a relational data representation, Platform for Virtual Data Integration (PVDI) promotes a bottom up approach, mapping the data infrastructure of an organization to a graph-based model.
It is important to realize that data infrastructure of an organization consists of both modelled data but also unstructured and generic that would produce outputs and results for decision-making. On the terms of data governance, it is difficult to understand these results, unless they are available in an intuitive to use manner.
In the era of the industrial revolution, modern organizations are collecting and managing an enormous wealth of data from different sources and applications. An ongoing huge hype for big data has been gained from academics and professionals, since big data analytics leads to valuable knowledge and promotion of innovative activity of enterprises and organizations, transforming economies in local, national and international level. Big Data are used in a variety of analytics projects, such as traditional BI, data exploration, data mining, etc. to provide a significant competitive advantage to the business. For example, a banking enterprise has data generated by a customer's banking transactions, credit card purchases, loans, web and mobile traffic, communication with the bank's call centre, and so on.
All this data is usually managed by different departments utilizing different systems and applications. The need to get data integrated under a common framework in order to make data more widely available for better control and governance has never been greater. Over the decades, with the increase of the data volume, as well as the format and variety we have entered the era of Big Data. The data integration has been demanding and can be seen as

constructing a data warehouse, or creating a virtual database (related terms: federated databases, mediators, multi-databases). While data warehousing was the way to go in the past - mainly due to the dominance of relational systems in data management - there are well-thought arguments to reconsider a virtual database approach.
The actual objective is to provide a solution as a graph based conceptual data model to provide data engineers and data scientists with a data infrastructure easy to understand and manage.

Scope

The scope is to develop a data canvas to capture the data infrastructure of an organization. The term "data infrastructure" involves not only data persistently stored in systems and files, but also generic data processing tasks that produce an output useful in decision-making.
With Data Canvas:

- Data Exchange/Data Sharing: Data and data processing tasks can quickly become part of the data canvas. For example, data engineers can select columns from an excel file and represent them as nodes of an entity in this graph. Data Canvas becomes the medium for data sharing in a standardized, collaborative, distributed manner rather than moving around files. The idea that enables this is the ability to map in the canvas data processing tasks – they acquire data semantics.
- Polyglot data preparation: Users can filter, aggregate and transform nodes creating new nodes. For example, a list of comments of a customer can become a list of numbers by applying a python program that computes the sentiment of a text on each comment.
- Visual Query Languages: Data scientists can easily pick attributes, transformed attributes, and nested attributes (along a hierarchy), in order to form a tabular view (a data frame) of the entity. This can serve as a stand-alone report or can serve as input to a ML algorithm. There is a solid theoretical framework for efficient query evaluation and optimization, based on an algebra we have developed over key-value stores.
- Model-specific Schemata on Demand: Using parts of the graph in a well-defined manner, a data-model specific schema can be defined, such as a star, or semi-structured schema. One can choose to execute a model-specific query on top of the derived schema, or materialize it. For example, a collection of documents can be rooted on CustID for an application (containing customer's transactions within the document), but another can define a collection of documents rooted on TransID.

The implementation of this project was based on the research idea of the published article of Mr. Damianos Chatziantoniou and MsVerena Kantere: Data Virtual Machines: Data-Driven Conceptual Modeling of Big Data Infrastructures. EDBT/ICDT Workshops 2020 [R8]

2. Theory

The example

Consider an example of credit card data that a bank desires to analyse in order to make important decisions on new products. These data reside in different sources of the bank system, other relational and other non- relational. For this project a predictive model should be designed and implemented taking into account the many possible variables (features) characterizing a customer and his credit card actions. The bank may have several data sources such as:

- A relational database containing customer's information and demographics;
- Flat files produces by statistical packages such as SAS and SPSS, containing data transformation and precomputed measures. These can be excel or csv files;
- From other web side data or other kind of databases (Mongo DB), json files from collections may be available;
- XML files from relational database or other sources may also contain metadata for a customer and his/her credit card transactions.

The aim is to equip the data analyst with a simple tool that enables fast and interactive experimentation by using features from multiple data sources, involving different data management systems and formats, in a systematic way. Such a tool would allow experimentation in an ad-hoc manner with multiple tabular views of customer-related data.
The goal was to create a Platform with a data canvas, where schema designers could easily map attributes and data scientists could simply define polyglot transformations over attributes and combine them into data frames. The evaluation of data frames should be efficient and based on a solid theoretical framework.

The conceptual model and the Data Canvas

The idea was quite simple: construct a graph around customer's id (the root node), where each node is an attribute (a domain of values) and an edge between nodes represents some computation that associates one or more values of an attribute to a customer's id (to be precise, a mapping from one domain to another domain). This is similar to an ER diagram, where all attributes are multi-valued and derived. Customer's id is also an attribute and becomes a node in this graph. The definition of a data frame is a tree rooted on the customer's id.

The PVDI provides a data working space, the Data Canvas, where users can easily (visually, graphically) represent both stored data and the output of data processing tasks, manipulate/transform data, and formulate data reports (queries). The data canvas describes entities and their attributes in a graphical way, much like the traditional Entity-Relationship Model (ER). A conceptual model like the ER is simple to understand, succinct, and depicts entities at a higher level. Users can easily pose constraints in this diagram, e.g. validity period of an attribute (i.e. for data governance), define visual queries by selecting specific attributes of an entity (i.e. for multi-source data framing) and quickly extend the schema of an entity by easily adding new attributes (i.e. for data sharing/data exchange in a standardized, agile manner). However, developing a conceptual model in the data canvas is the reverse process of the one followed in ER design: while ER uses a top-down approach, where real-world entities and their relationships are depicted and utilized in a process that leads to a relational (in most cases) data representation, the data canvas promotes a bottom up approach, from data model-specific implementations to an agile conceptual model. In other words, in ER we start from

the conceptual model and conclude to a model-specific implementation, in data canvas we map model-specific implementations to a conceptual model. Figure 1 shows Data Canvas modelling vs ER modelling. Multi-databases, virtual databases, mediators, federated databases are related concepts in data management literature.

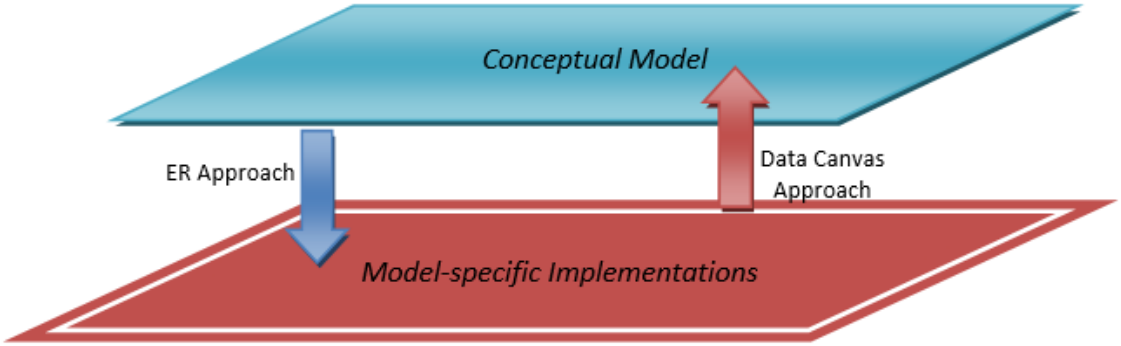


Figure 1: Data Canvas modelling vs. ER modelling

The key idea in the Data Canvas is to make it easy to add an attribute to an entity from a variety of data sources (relational databases, flat files, excel files, NoSQL, etc.) For a customer entity, examples of attributes include his age, gender and income, but also his emails, images, locations and transactions. An attribute of an entity could have one or more values – for example, the age of a customer is a single value, but the emails of a customer can be many – in ER theory these are called multi-valued attributes. In ER-theory attributes can be standard or derived. A derived attribute is an attribute where its value is produced by some computational process, i.e. there exists a process that attaches one or more values to the entity. In Data Canvas, since we map *existing* data to entities, we can only have derived attributes. For example, the query "**SELECT custID, age FROM Customers**" can be used to bind an age to the customer entity (using the primary key of the entity, **custID**). Figure 2 shows this idea. The computational process that "defines" the attribute is attached to the edge.

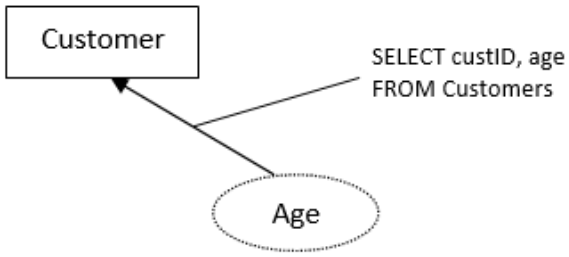


Figure 2: A derived attribute in the Data Canvas and its defining process

Actually, this is the gist of our model: computations that associate (map) values to an entity are labeled (named) as attributes of that entity. This is very powerful, because you can semantically represent any data manipulation task onto a conceptual model. Examples involve the SQL statement mentioned above, but also a MongoDB query, a Cypher query, programs that read from a flat or an excel file, even programs such as the one mentioned in the introduction that assigns a churn probability to each customer. The only requirement is to map one or more values to an entity, i.e. to have a two-column output. An important observation to

make is that this computation can be intra- or inter-organization. Figure 3 shows additional attributes for the customer entity (for simplicity we draw attributes with a solid line rather than a dashed line).

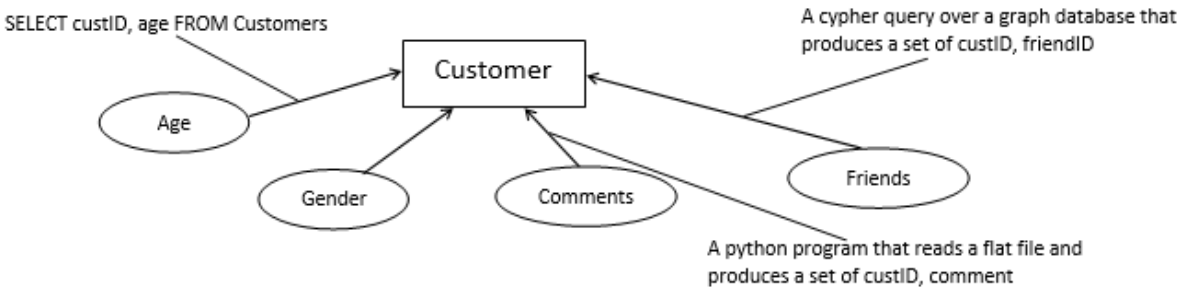


Figure 3: Customer entity with several attributes

So we have a conceptual model, where all attributes are multi-valued and derived. We assume that all entities have a primary key (a quite realistic assumption in most real-life implementations), so an entity can be represented by its primary key, which is also an attribute. Figure 4 represents the customer entity as an attribute.

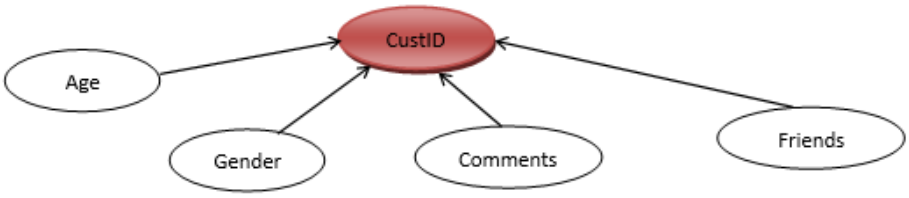


Figure 4: Customer entity represented through its primary key

The transactions of a customer (consisting of transIDs) is another attribute (multi-valued) of the entity customer, but at the same time is an entity itself, with its own set of attributes, which means that there is no need for relationships, as in the traditional ER diagram (Figure 5). This observation is important because it simplifies the conceptual model to a graph (we only have attributes), which opens up a wide range of opportunities, as explained shortly.

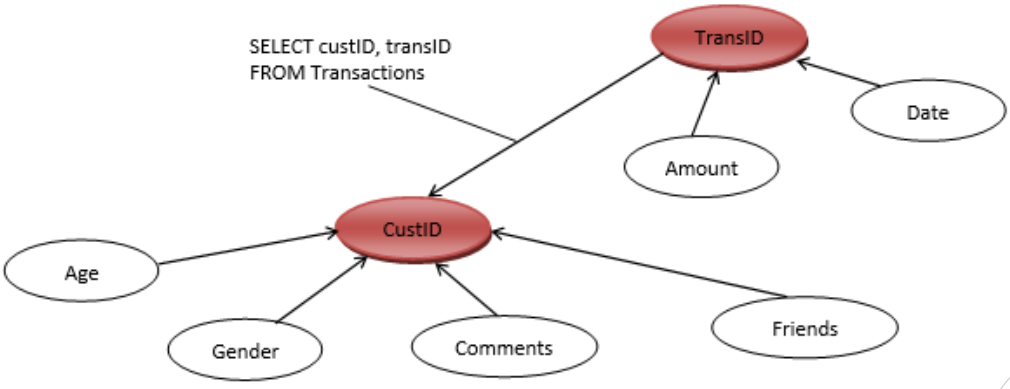


Figure 5: Customer and Transaction entities

Finally, let us consider query "SELECT custID, age FROM Customers" of Figure 3. While this query maps an age to a customer (to a custID to be precise), it also maps one or more custIDs to a specific age value (reverse mapping). In other words, a computation with an output {(u,v): u∈U, v∈V} (multi-set semantics) provides two mappings, one from U to V and one from V to U. This means that edges in a Data Canvas graph are bidirectional (Figure 6). In that respect, all nodes in this graph are symmetrical, i.e. there are no primary keys. However, one can consider a node with degree > 1 as a primary key.

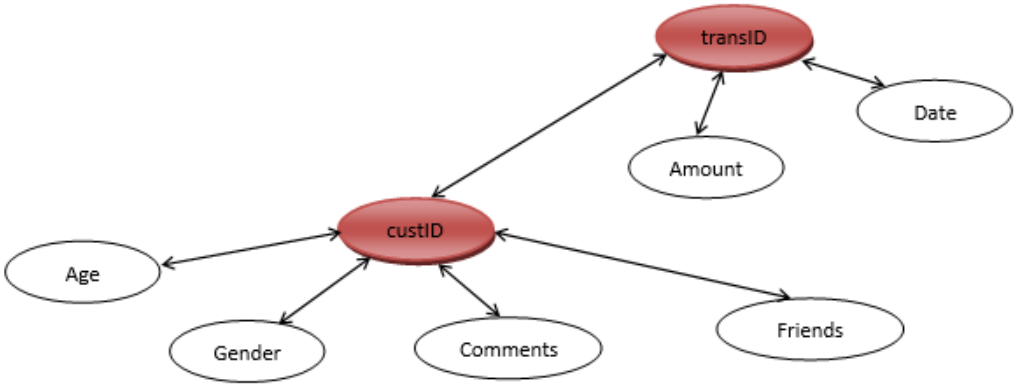


Figure 6: Customer and Transaction entities, a running example

3. Methods

Designing the Platform for Virtual Data Integration, the first question that needs to be answered is how the data are going to be stored. The data model is quite important on the overall architecture of the tool and defines the implementation.

A key-value store, or key-value database, is a type of data storage that stores data as a set of unique identifiers, each of which have an associated value. [R6] It is quite popular in DB-Engines Ranking, well known with database models as Redis. It is similar to a dictionary or a map data structure. Key-value stores can be considered as the most primary and the simplest version of all the databases.

A key-value store can be very fast for read and write operation. It can be very flexible as more data may be generated without traditional structures. [R5]

Requirements

Key – Lists Pairs

The basic requirement is to extend the key-value pairs into key-list pairs.
Assume a collection of (key, list) pairs, i.e. the value is a list of values, namely strings. Assume that key is a string as well. Let us call such a collection a Key-List Store (KL Store). This is a special case of a multi-map data structure, where several values are mapped to a key. A KL Store is the following:

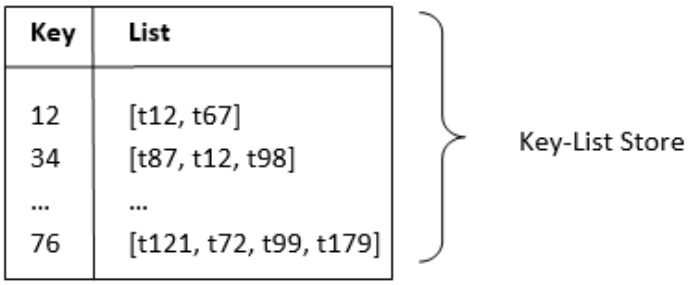


Figure 7: KL Store example

Assume two domains of values D1 and D2:

D1 = {all possible customer ids}, D2 = {all possible transaction ids}

Assume that there is a process P that generates a collection of (value1, value2) pairs

S = {(u,v): u ∈ D1 , v ∈ D2}

Examples of such processes:
- SELECT custID, transID FROM SALES
- Reading a CSV file and getting for each line forming a pair using columns i and j.
- Running any program that produces a stream of pairs of values

Given a collection S described as above, one can define two KL Stores, KL1(S) and KL2(S) as follows:

KL1(S) = {(x, Lx), ∀x ∈ U = {u: (u,v) ∈ S}, Lx = the list of values v, such that (x,v) ∈ S}
KL2(S) = {(x, Lx), ∀x ∈ V = {v: (u,v) ∈ S}, Lx = the list of values u, such that (u,x) ∈ S}

Schema

A graph is going is created in the following manner:

For each domain, Di a node Di is created. If there are two domains (nodes) D1 and D2 and a process P that generates a collection of (v1, v2) pairs, S = {(u,v): u ∈ D1 , v ∈ D2} then we create two edges, D1→D2 and D2→D1 labelled with KL1(S) and KL2(S) respectively.

Graph G is the schema corresponding to a collection of domains and a collection of processes associating values between two domains. An example of the schema may be find below:

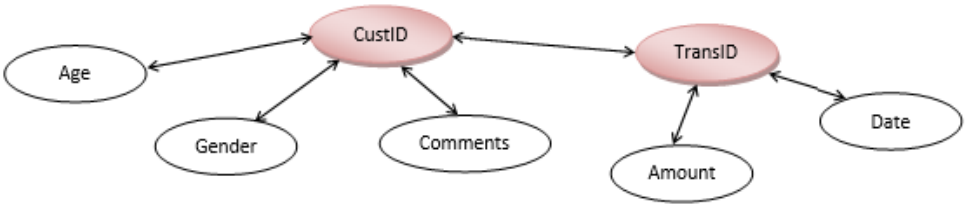


Figure 8: Example of the schema

All edges correspond to processes that generate a collection of (value1, value2) pairs. The different colour (CustId and TransID) on some nodes is just to identify nodes as "primary keys". The defining property for a node x to be a "primary key" is:

Number of neighbours(x) > 1.

Evaluation Algorithm

A schema on Demand may be a json file produced by a Mongo DB collection.
Given the tree on Figure 8: Example of the schema, we need an evaluation algorithm that generates a comma-separated collection of json documents having as root element the root of the tree. For example, the json file below:

```
{
   CustID: "234",
   Age: "48",
   Gender: "F",
   Comments: ["hello world","that was a good hotel","piss off"],
   TransID_s: [
{
```

```
    TransID: "A231-091",
    Amount: "34.56",
    Date: "23/4/2018"
},
{
    TransID: "A565-075",
    Amount: "129.42",
    Date: "23/4/2018"
}
    ]
},
…
```

Assume a node A and n of its children, A1, A2, …, An. We assume the presence of a function **combineKeys(A, A1, A2, …, An)**, which returns a list of the union of the keys for A that can be found in the association between A and Ai, for each i, i.e. in the edges A→Ai (KL stores AA1, AA2,…, AAn).

Furthermore, for an edge A→B (KL store AB), assume a function **retrieveList(k, A, B)** that returns the list of key k in A→B.

Input: A tree rooted on node A

```
main(A) {
        K ← combineKeys (A, A1, A2, …, An);
        for each k in K {
          if (not the first k in K) println ",";
          println "{";
          println "$A:\"$k\",";
          for each child B of A {
            eval(k, A, B);
            if (not the last child) println ",";
          }
          print "}";
        }
      }

eval(k, A, B) {
  L = retrieveList (k, A, B);
  if (B has no children) {
    print("$B:");
    if (length(L)==1)
      println("\"$L[1]$\"");
    else
      println("[$L]");
  }
  else {
    println "$B"+"_s: [";
    for each l in L {

      if (not the first l in L) println ",";
      println "{";
      println "$B:\"$l\",";
      for each chievald C of B {
        eval(k, B, C);
        if (not the last child) println ",";
      }
      print "}";
    }
    println "]";
  }
```


Table 1: Combine Keys Algorithm

Data sources

The input data sources will be collected in a dictionary or better in a configuration file describing each data source. The file will be in Json or XML format.
The basic idea is each sources to be described by an ID, its type and the connection string which may differ based on the type of data source. For example for a csv file may be the file path, whereas for an sql source may be the connection to the database.

```
<datasources Repository=”SomeName”>
        <datasource name=”DSName” id=”someID” type=”db/excel/csv”>
        <dbconnect>
                <username> username </username>
                <password> username </password>
                <request string> IPaddress <request string>
                <database> DBname </database>
        </dbconnect>
        <filename> someFileName </filename>
        <path> somePath </path>
        <delimiter> someDelimiter </delimiter>
        </datasource>
</datasources>
```

Architecture

Based on the requirements listed in the previous section (Requirements) the architecture of PVDI was created to reflect all the work from the data source to the final output.

Each layer of the architecture is implemented in a separate function on the programme.

1. Data sources. The data sources may be excel or csv files, relational databases and json files. The programme will “understand” each data source by reading from the configuration file. The data then will be loaded.
2. The schema is represented as a graph. Each node is an attribute with a name, i.e. CustID for Customer, TransID for Transaction etc. Each edge is the relationship between two nodes and contains the data source ID, the Type and connection string as in the example below:

   "1;csv;Transaction,CreditCard"
3. All the data will be processed in a following layer, in a programme that will read the graph and will extract the data source and the connection strings. i.e. (ds; 3,5) where ds is a csv file and 3,5 are the columns 3 and 5 that we are interested in.
4. The programme will create a key-list structure using Redis in the format (K,L)→ (Key, [l1,l2,l3…ln]).

   For example if from the programme two values are produced each time:
   A[(1,8), (1,9), (2,9), (2,10), (3,8), (4,9)..], then two collections may be created with a

key and a value as a form of a list.
S1: {(1, [8, 9]), (2, [9, 10]), (3, [8]), (4, [9])}
S2: {(8, [1, 3]), (9, [1, 2, 4]), (10, [2])}

This means that the user may choose each time another key in order to collect all the related values in a list.

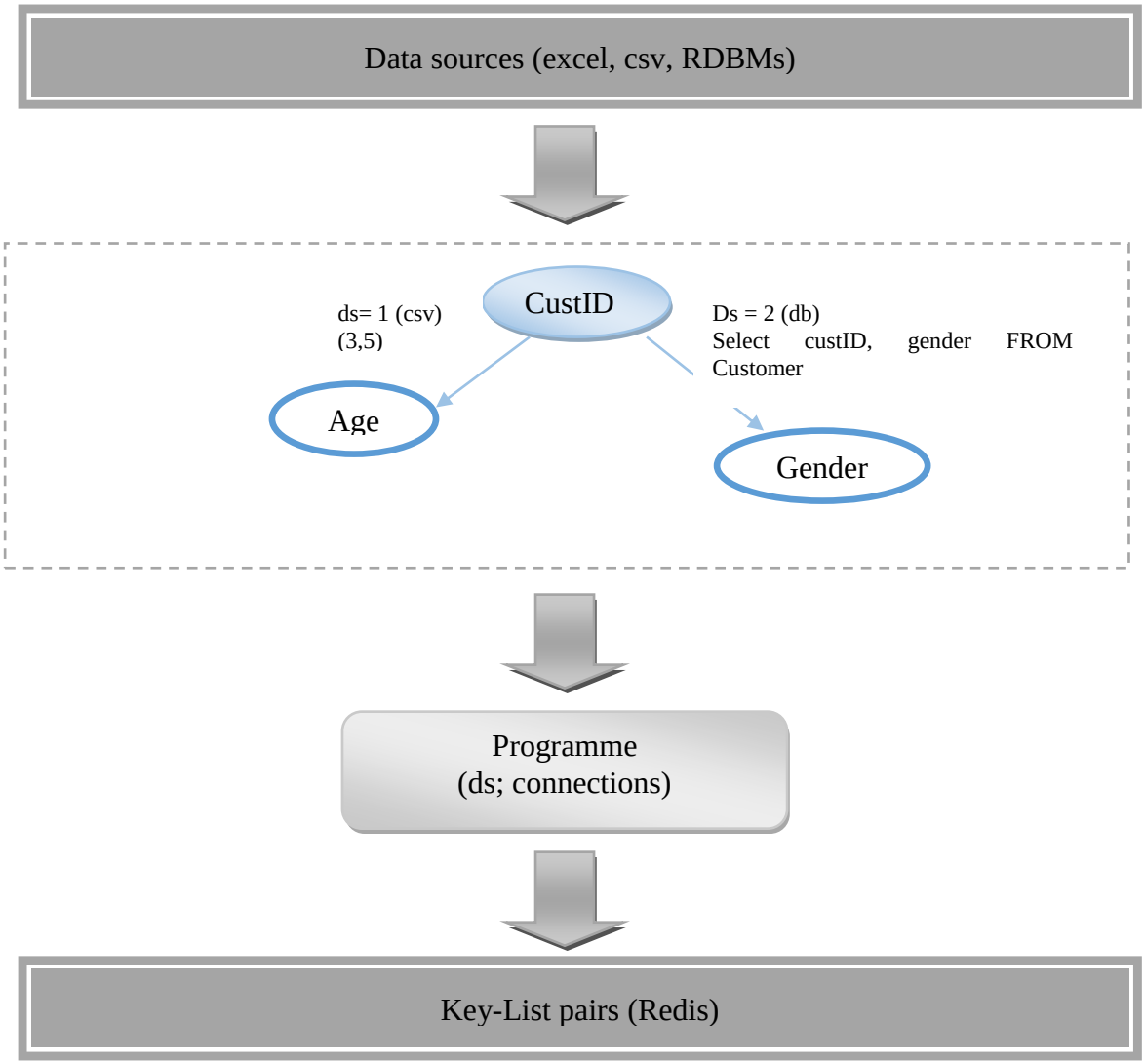


Figure 9: Architecture of PVDI

Implementation

Languages

The core development of PVDI was implemented with **Python 3.6.**

Cypher was the graph query language that was used to query the graph database**.**
Finally, **Redis** was used for the creation of the key – list pairs.

Tools

The list of tools that were used may be found below:

Tool	Description	Version
Spyder [1] (via Anaconda navigator)	Python IDE with editing, testing and debugging	v3.2.8
Neo4j Desktop[2]	Graph Database	v1.2.1
SQL Server Management Studio[3]	Microsoft SQL Server	v18.2
Redis Client[4]		
Notepad++[5]	Flat file editor	v7.8.2

Table 2: List of Tool used

The development was done in computers with the following characteristics:

- Processor: Intel i5, 2.20 GHz
- RAM: 8.00 GB
- System 64-bit
- Software: Windows 10 Home

Functions

This section describes in detail the creation of each function and how each one is used.

- **CreateGraph.py**
  This python file is the first that should be run for graph's creation. In this file, the command line for graph's creation is created.

- **ConnectToNeo4J.py**
  This file consists of function *createGraph*. In this function, neo4j is called to create the graph with the predefined nodes and relationships that have been selected from user.

- **Main.py**
  This python file is the next file that should be run from command line in order to create the key-value pairs in Redis. From command line, function *callNeo4jParameters* is called given as arguments on command line two parameters, nodeA and nodeB that define the key and value for Redis. First, given the two arguments, neo4j is called to give all relationships that each node has. After detecting the relationships for each node, function *ReadFromFilesFunctions* from file readFromNeo4jInput and *create_KV_Lists* from file Connect_To_Redis are called

[1] https://www.anaconda.com/distribution/
[2] https://neo4j.com/download/
[3] https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15
[4] https://redis.io/download
[5] https://notepad-plus-plus.org/downloads/

serially to create the key-value pairs.

- **ReadFromFilesFunctions.py**
  In this python file, first the config json file is called to load parameters that define data sources and the configuration for these. Then, function *readFromNeo4jInput* is called that defines for nodes NodeA and NodeB the source for each node. Based on type of data source different function is called.
    - *readFromCsv* is the function that reads the path for data source and the columns that should be read from csv.
    - *readFromExcel* is the function that reads the path for data source and the columns that should be read from excel.
    - *readFromDatabase* is the function that reads the path for data source and the columns that should be read from database that information is stored.
    - *readFromJson* is the function that reads the path for data source and the elements that should be read from json document.

- **Connect_To_Redis.py**
  In this python file, connection to Redis is established. Function *create_KV_Lists* is called to create the Redis lists based on nodes A and B that are defined in Main function call.

- **Combine_Keys.py**
  In this python file, a json document which is created for a selected subtree from created graph, could be exported in .txt file. The following functions are called:

    - Function *callRoot* is the main function that is called to create json. Parameter that is passed in function *callRoot* is the node that is characterized as root for the subtree. Then from graph user can choose which parent nodes he/she wants to add to subtree.
    - Function *findKeys* is called with parameters these selected nodes in order to create the list with the nodes that are selected as parents.
    - Function *callNeo4ForJsonParameters* is called afterwards to find based on the list of parent nodes, all children of these nodes. Neo4j is called with cypher query to find the parents with their children nodes.
    - Function *combineKeys* is called with parameter the results of function *callNeo4ForJsonParameters.* Assume a node A and n of its children, A1, A2, …, An. We assume the function **combineKeys (A, A1, A2, …, An)**, which returns a *list* of the union of the keys for A that can be found in the association between A and Ai, for each i.
    - Because or the subtree is undirected, some relationships were created more than one time. So, function *remove_Duplicates* is called to remove some duplicate keys.
    - Afterwards, function *evaluate* is called to find the key-value pairs for each key.
    - Function *retrieveList* is called to find the values of a list in Redis based on parameters and function *findChildrenKeys* to find the keys of each child of parent node.
    - In function *evaluate* the creation of json document based on the following algorithm is implemented.

Test Data

The Test Data that were used for the development of this project were found online in the Kaggle [R7] website.

The data are dummy and do not reflect any real situation of person. They were transformed in several needed types for this project. All the test data may be found in APPENDIX. Though below it is presented in detail how the data were manipulated.
CardBase.csv: This table has credit card level information. Contains 4 columns and 500 rows. The columns are the card number, the card family, the credit limit and the customer ID. The same file may be found in excel version with name **CreditCardData.excel**.
TranscactionBase.csv: This sheet has transaction information done on credit cards. Contains 4 columns and 10.000 rows. The columns are Transaction ID, Transaction Value an Transaction Segment.
Transcactions.csv: This sheet has transaction information related to customer. Contains 2 columns and 10.000 rows for Transaction ID and Customer ID.
Customer_data.json: This file contains customer related data in json format.
Transcaction_data.json: This file contains transactions in json format.

Finally, a relational database was created with customer data (Personal information: First/Last Name, Age, Gender, Occupation and the Customer ID). The insert DB query may be found in Table 14: Insert DB query.

Installation

In this section, it is presented a step by step installation from scratch. It may be considered as a user manual for a data scientist or developer.

1. Prerequisites
    a. All the tools mentioned in Tools section should be installed and running based on their installation instructions.
    b. The Test data mentioned in Test Data section should be all saved in the same path. Assume the path: C:\. Save all six files (provided in the deliverable folder) in the same path. Additionally with the Test Data, the config.json should be saved in C:\.

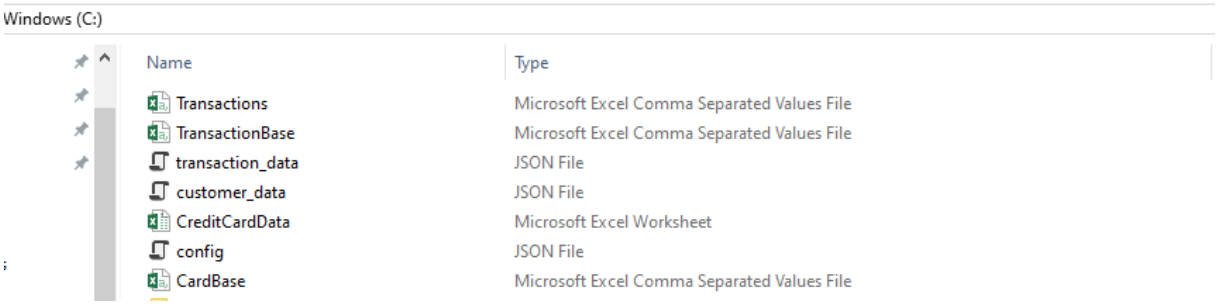


Figure 9: Test Data path

    c. Neo4j graph Database should be running. A new Project must be created along with a new graph. To do so follow the next steps:
        - Open Neo4J.
        - Create a new Project by adding on New on the top of the page. Rename your project accordingly.

- Select the newly created project and add a new graph. Name your graph accordingly.

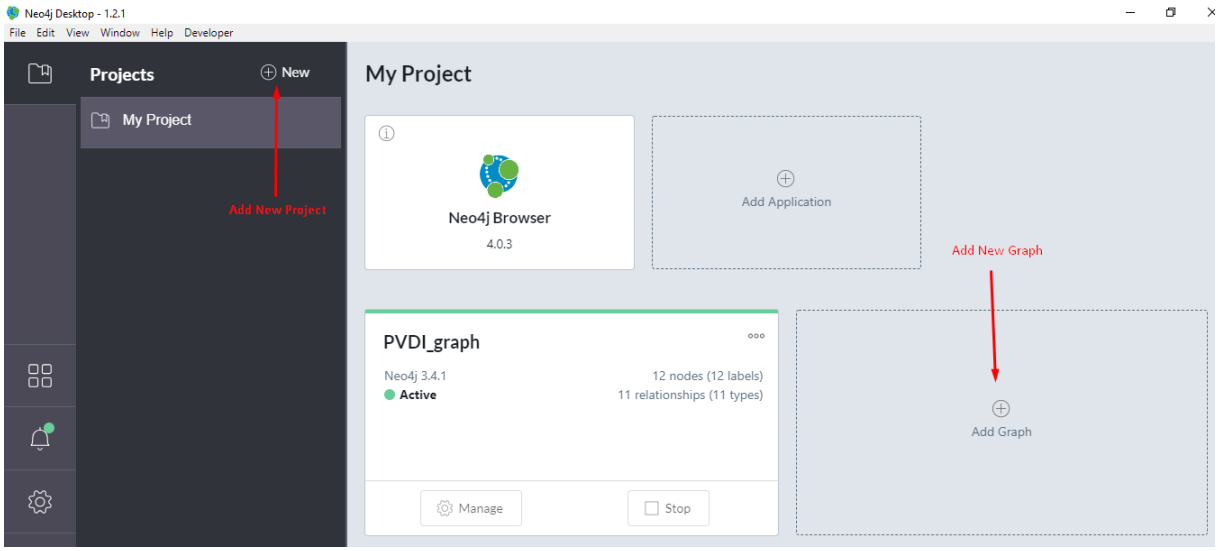


Figure 10: Neo4J - New Project & Graph

- Click on Manage and press on start symbol to start the new graph. The HTTP Port should be 7474. In any other case, the port should be configured manually in the code. (Main.py lines15 and 16, CreateGraph.py lines 10 and 11. Save.)

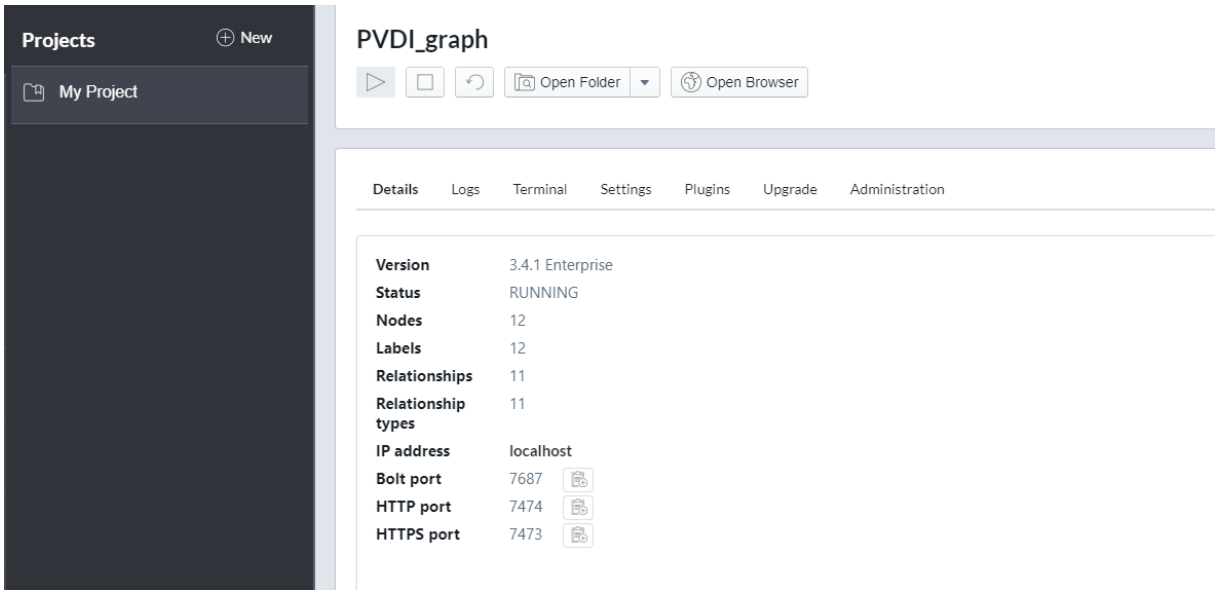


Figure 11: Neo4J HTTP Port

d. Customer table should be created to RDBMS prior to installation.
   - Open Microsoft SQL Server Management Studio.
   - Create a new Database named PVDI.
   - Click on New Query.
   - Copy/Paste insert query as found in APPENDIX, Table 14: Insert DB query.
   - Click execute.

e. The configuration file should be updated it with the relevant server connection credentials.

2. Open Anaconda Prompt.
   Navigate to the path where the code is saved. Assume that the code is save in in a folder named PVDI in Documents. Execute:
   >cd Documents\PVDI
   >dir

```
(base) C:\Users\Yrouli>cd Documents\PVDI

(base) C:\Users\Yrouli\Documents\PVDI>dir
 Volume in drive C is Windows
 Volume Serial Number is 6185-6E5E

 Directory of C:\Users\Yrouli\Documents\PVDI

08/03/2020  19:58    <DIR>          .
08/03/2020  19:58    <DIR>          ..
02/03/2020  10:26             4,459 Combine_Keys.py
02/03/2020  10:26             2,507 ConnectToNeo4J.py
02/03/2020  10:26               908 Connect_To_Redis.py
08/03/2020  18:46               453 CreateGraph.py
08/03/2020  18:48             1,134 Main.py
02/03/2020  10:27             2,372 ReadFromFilesFunctions.py
               6 File(s)         11,833 bytes
               2 Dir(s)  240,061,542,400 bytes free
```

Figure 12: Anaconda Prompt - Python files

3. Execute the following command:
   >python CreateGraph.py
   To create the graph in Neo4J.

4. Execute manually in Neo4j graph DB, all the properties to create the relationships in the graph. The properties are found in Table 15: Neo4j Properties.

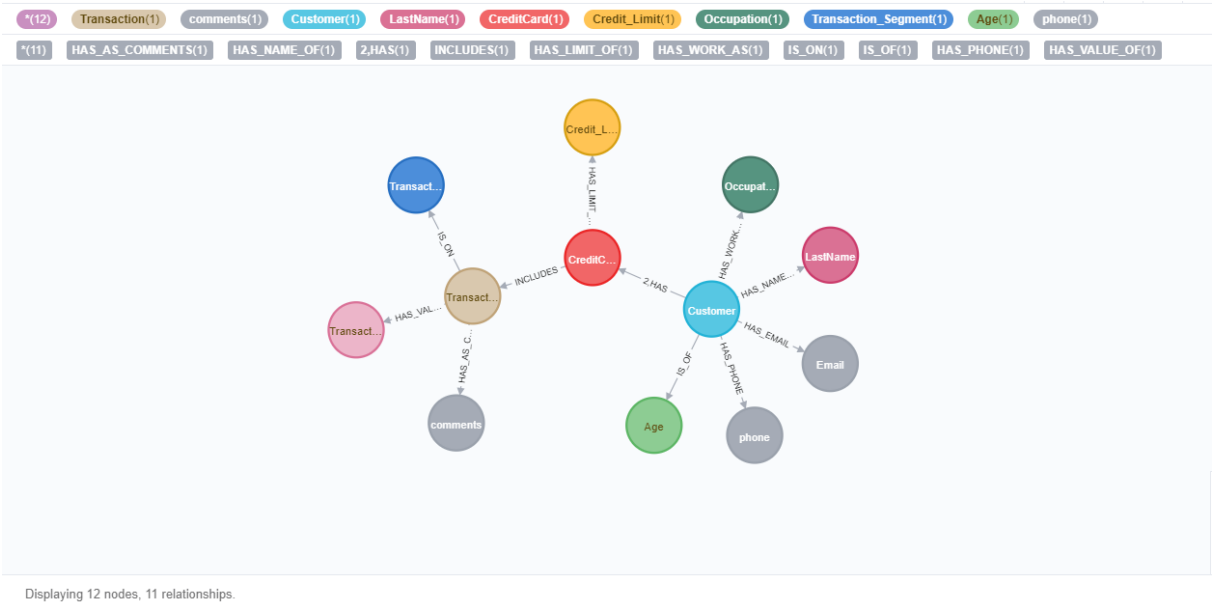


Figure 13: Neo4j graph

5. Navigate to the file path where Redis is saved. Assume that you have saved the zip in Programme files. Navigate to C:\Program Files\Redis and double click on redis-server and redis-cli to start the redis server and the redis client.

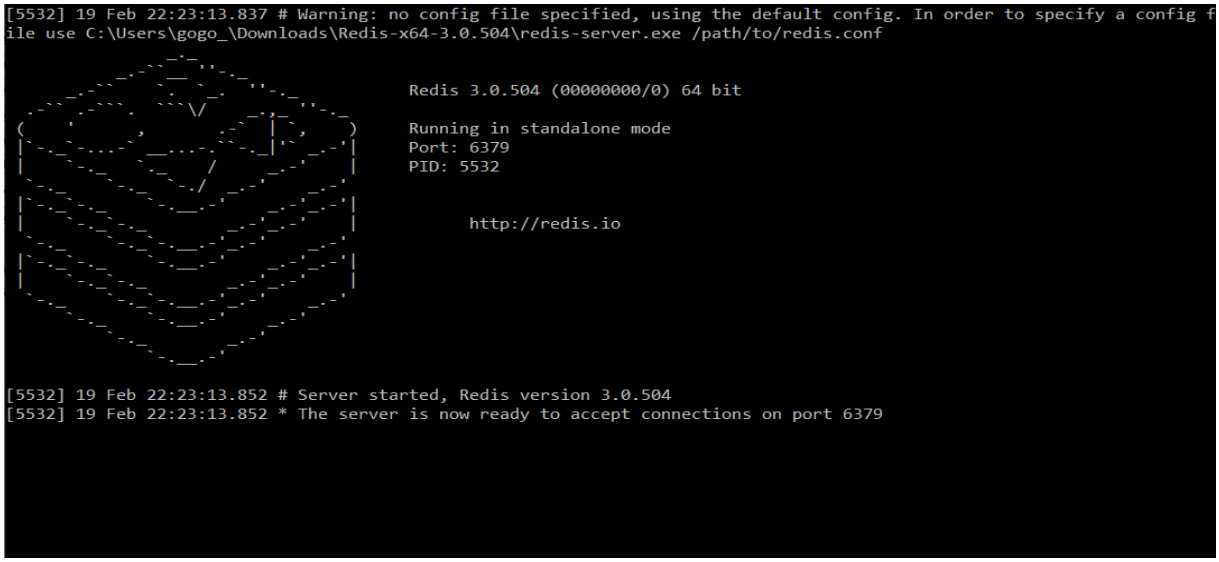


Figure 14: Redis server start

6. To run all the Key-List pairs execute the following command using each time a different pair from the table
   >python Main.py Customer CreditCard

Execute the above command with all the pairs from the table below (as they mentioned).

Customer Age
Customer LastName
Customer Email
Customer Occupation
Customer CreditCard
Customer phone
CreditCard Credit_Limit
CreditCard Transaction
Transaction Transaction_Segment
Transaction comments
Transaction Transaction_Value

Table 3: Pairs

Execute all the reversed pairs as following:
>python Main.py CreditCard Customer

7. To create json document the following execution should be run in command line:
   >python Combine_Keys.py nodeX where nodeX the selected root node from user.

*** Note: In case of re-installation and in order all the above steps to run in clean environment, the below steps should be followed:

a. Delete the Neo4J graph. Execute in Neo4j:
   MATCH (n)
   DETACH DELETE n
b. Delete the Customer table from the RDBMS. Execute in SQL Management Studio:
   DROP TABLE dbo.Customer;
c. Clear Redis. Execute in Redis Client:
   FLUSHALL

After the above three steps, a clean installation may be performed.

4. Testing and Results

To test the validity of the code, the installation process and that the project meets all the requirements, a set of tests was executed.

Test Case 1: Graph DB Configuration	
Purpose	Test the installation of Neo4J.
Prerequisites	Neo4J to be installed
Steps	1. Execute step three of the installation steps and verify that 12 new nodes have been created.
	2. Execute step four of the installation steps, by running one by one all the properties. Verify that each edge on the graph has a relationship.
	3. Mouse hover on the edge between two nodes. Verify that the property on the edge has data source ID, the data source type and the two connecting nodes.
Expected results	1. A new graph with 12 nodes has been created. The nodes are: Transaction, Transaction_Segment, Transaction_Value, comments, CreditCard, Credit_Limit, Customer, Occupation, LastName, phone, Age and Email.
	2. The graph contains 12 nodes and 11 edges, each one with one relationships.
	3. The edge connecting Customer and Credit Card has "2;excel; CreditCard Customer"

Table 4: Test Case 1- Graph DB configuration

Test Case 2: Redis	
Purpose	Test whether Redis is up and running.
Prerequisites	Redis server and Redis client to be installed.
Steps	1. Open Redis Client.
	2. Execute the command: ping.
	3. Execute "set MyKey SomeValue".
	4. Execute "get MyKey"
Expected results	1. Redis client opens with the IP: "redis 127.0.0.1:6379".
	2. The result is PONG.
	3. The result is OK.
	4. The result is "SomeValue"

Table 5: Test Case 2 – Redis

Test Case 3: Key – List Pairs	
Purpose	Test the creation of KL pairs.
Prerequisites	The installation process should be successful. Important is that graph DB will be started and the RDBMS database credentials are updated in the configuration file.
Steps	1. Execute the following command for the first pair of nodes as depicted in Table 3: Pairs: python Main.py Customer Age. Verify that the result is present is as Node A_Node B_Key:['Value']
	2. Verify the values are correct by checking the Database.
	3. Execute the same command by reversing the Nodes: python

	Main.py Age Customer. Verify that key is the Age in each pair.
Expected results	1. The result is as expected: Customer_Age_CC67088: ['32'] Where key is the Customer CC67088 with age 32.
	2. The values are matching the database data.
	3. The pairs have key the age and value the Customer as following: Age_Customer_32: ['CC67088']

Table 6: Test Case 3 Key - Lists Pairs

Test Case 4: Key – List Pairs	
Purpose	Test the creation of KL pairs.
Prerequisites	The installation process should be successful.
Steps	1. Execute the command: python Main.py CreditCard Transaction. Verify the KL by checking the data in the Transactions.csv file.
	2. Verify that with the Credit Card '1947-8602-1695-7503' there are 20 different transactions.
	3. Repeat the above process for another Credit Card. (i.e. 6699-2639-4522-6219, 9391-7086-9113-4893 etc.).
	4. Execute the command python Main.py Transaction CreditCard
Expected results	1. The result is as expected.
	2. By filtering the Transactions.csv file with the Credit Card '1947-8602-1695-7503', the result are 20 transactions.
	3. There are 15 and 12 transactions respectively.
	4. The pairs have key the Transaction and value the CreditCard as following: Transaction_CreditCard_CTID64730271: ['1947-8602-1695-7503'].

Table 7: Test Case 4 Key - List Pairs

For consistency, reasons all the possible KL pairs were tested with the nodes as depicted in Table 3: Pairs. The results were verified from the respective data sources as given in the deliverable folder.

5. Discussion

The results of the previous sections demonstrate two things. First, the initial requirements are met with the creation of KL pair's structure with data extracted from several different sources. Second, the Data Canvas, which was created as a graph representation of the data manipulated.
The PVDI is not a complex application with many clicks for making results or in depth understanding of composite data structures and formats. It is a simple, user-friendly BI tool that may help every user to combine information or create reports without any complex manner.
Nowadays there are many other data virtualization tools that used by several organizations to integrate data and give a more simple and unified view to the user. Comparing the implementation of this project with these tools, similarities may be found on the scope and definition of the back end of a data integration tool. Even though this is not a replica of any other commercial tool, our results suggest that a structure like data canvas may be used is several other applications. The technology used behind, the Redis KV pair structure, is one of the top KV stores and can be extended to provide more useful features to the user. Additionally, demonstrates a great performance on the storing of values in lists with a bottom up approach, mapping the data infrastructure to the data canvas model.
Data virtualization is not only for extracting data but also for giving more flexible and improved operations in the Business Intelligence world. It provides a unique approach to data management in many systems providing agility for adding, removing or changing data that are involved into systems. Quite important part is the data governance with main areas the availability of the data, the usability and the consistency. With virtual data integration, the data management may be processed more effectively along with the data privacy issues of GDPR.
The speed and ease of a query processing is quite important too. Data are always accurate as they are up to date every millisecond, as it allows to organizations to make better decisions. From a business perspective, getting integrated information faster has resulted in numerous benefits including better customer retention and revenue growth, faster product launches, reduced risk, etc. As a result, the reports are more accurate with quality information. It enables business leaders to make better decisions and improves business performance, without many costs due to the flexibility of modern systems. The risks for business changes are decreased while chief risk officers can improve campaign performance and chief information officers can speed up the business outcomes at a lower cost. Furthermore, by delivering data real time, business users could access the most current data during regular business operations. Virtual data integration not only has benefits for businesses but for customers too. Customer's satisfaction and time optimization are some of the benefits that data virtualization has.
Businesses that will use this tool could decline the risk of using untrusted data and the productivity can be increased as data can be accessed in a variety of ways depending on what it is used for. Some examples of business roles that could use this tool are the following: Analytics Leaders: Data canvas simplifies and speeds access to the data that you need to power the analytics tasks. Business Analysts and Data Scientists through this tool could provide instant access to all the data that they need and the way they want them. CIO, Data Engineering Leaders and CTOs use data canvas to add flexibility on data management strategies and architecture.

6. Conclusions

In summary, this paper argued the virtual data integration with the creation of a platform that will present a graph representation of the data to the user. The data canvas, as it we called it above, is an easy way to map attributes to entities from several data sources (excel, csv, relational databases, NoSQL etc). To do so, we created KL pair's structures that associated values with a set of identifiers, the keys. The KL pairs were extracted from a graph of entities with specific relationships pointing the data source and each type. By introducing the KL pairs store in the implementation of the PVDI we indicated very fast read and write operations in a more flexible way, judging by the data generated from non-traditional structures.

In conclusion, as the data sources grow and become more and more complex inside each business, the old techniques of manipulating the data seem to become obsolete. The virtual data integration is an approach were data can be abstracted from several source systems allowing a virtual data layer to process and consolidate them. This greatly streamlines an organization's ability to access and consume its data while saving time and money.

As also discussed above, all future developments should point to virtualization and strongly consider the combination of several BI options. The possibility to use the traditional data warehouses for example, for federating the data and the virtualization for cutting costs and getting quicker results will be desirable for future projects.

7. References

[R1] Data Integration: https://en.wikipedia.org/wiki/Data_integration
[R2] Data Virtualization https://en.wikipedia.org/wiki/Data_virtualization
[R3] Accur8 Software: https://accur8software.com/data-virtualization-a-superior-data-integration-solution/
[R4] Article in Journal of Computer Information Systems-May 2018, V-DIF: Virtual Data Integration Framework. [https://www.researchgate.net/publication/329567828_V-DIF_Virtual_Data_Integration_Framework]
[R5] Key-Value store advantages, [https://hazelcast.com/glossary/key-value-store/]
[R6]https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46
[R7] Kaggle: https://www.kaggle.com/ananta/credit-card-data
[R8] Damianos Chatziantoniou, Verena Kantere: Data Virtual Machines: Data-Driven Conceptual Modeling of Big Data Infrastructures. EDBT/ICDT Workshops 2020

8. APPENDIX

All the code and the Test Data included in the deliverable CD may be found below.

ReadFromFilesFunctions.py

```
#Libraries
import pyodbc
import pandas as pd
from pandas import ExcelWriter
from pandas import ExcelFile
import json
import csv

#The file path of the configurarion file
config='C:/config.json'

#Open the config file and load the data sources
with open(config) as f:
   data = json.load(f)
json_string = json.dumps(data)
resp = json.loads(json_string)

#Function that Neo4j calls to read data from data sources
#Based on type of data source different function is called
def readFromNeo4jInput(Source, Type, QueryParameters, nodeA, nodeB):
  if Type == 'csv':
    mylist = QueryParameters.split(',')
    if(nodeA == mylist[0]):
       columns2 = [mylist[0], mylist[1]]
    else:
       columns2 = [mylist[1], mylist[0]]
    return(readFromCsv(Source,columns2))
  elif Type == 'sql':
     return readFromDatabase(Source,QueryParameters,nodeA, nodeB)
  elif Type == 'excel':
     mylist = QueryParameters.split(',')
     c1= "+ mylist[0] +"
     c2= "+ mylist[1] +"
     if(nodeA == c1):
        print(c1)
        columns2 = [c1, c2]
     else:
       columns2 = [c2, c1]
     return readFromExcel(Source,columns2)
  elif Type == 'json':
     mylist = QueryParameters.split(',')
     c1= "+ mylist[0] +"
     c2= "+ mylist[1] +"
     if(nodeA == mylist[0]):
        columns2 = [c1, c2]
     else:
```

```
        columns2 = [c2, c1]
      return readFromJson(Source,columns2)
  else:
    print('Other')

#Function read from csv
def readFromCsv(K, columns):
    a = resp['datasource'][K]['Connection']
    csv_df = pd.read_csv(a)
    return(csv_df[columns])

#Function read from excel file
def readFromExcel(K, columns):
    excel_load  = resp['datasource'][K]['Connection']
    df = pd.read_excel(excel_load, index_col=None, na_values=['NA'],squeeze=True)
    new_df = df[columns]
    return(new_df)

#Function read from database
def readFromDatabase(K, query,nodeA, nodeB):
    conn  = pyodbc.connect(resp['datasource'][K]['Connection'])
    mylist = query.split('-')
    query = mylist[0]+ " "+ nodeA+ ","+ nodeB + " "+mylist[3]
    action_records = pd.read_sql(query, conn)
    db_df = pd.DataFrame(action_records)
    return(db_df)

#Function read from json
def readFromJson(K, columns):
    json_load = resp['datasource'][K]['Connection']
    json_df = pd.read_json(json_load)
    json_df_2 = json_df[columns]
    return(json_df_2)
```

Table 8: Read from files function

CreateGraph.py

```
#Libraries
from py2neo import authenticate,Graph
from ConnectToNeo4J  import createGraph
from py2neo.packages.httpstream import http
http.socket_timeout = 9999

#Neo4j connection configurarion
authenticate("localhost:7474", "neo4j", "123")
graph = Graph("http://localhost:7474/db/data/")
#Create graph databaseon --call from command line to create the graph

#Main Functi
if __name__ == "__main__":
    print("Start Create graph...")
```

```
    createGraph(graph)
    print("End Create graph")
```

Table 9: Create Graph Function

ConnectToNeo4J.ph

```
#Libraries
import pandas as pd
from py2neo import authenticate,Graph
from py2neo import Node, Relationship
from py2neo.packages.httpstream import http
http.socket_timeout = 9999
#Create graph function
def createGraph(graph):
    #Create Nodes
    print("Create Neo4j Nodes")
    Transaction_ID = Node("Transaction", name="Transaction")
    Transaction_Value = Node("Transaction_Value", name="Transaction_Value")
    Transaction_Segment = Node("Transaction_Segment", name="Transaction_Segment")
    Comments = Node("comments", name="comments")
    Credit_Card_ID = Node("CreditCard", name="CreditCard")
    Credit_Limit =  Node("Credit_Limit", name="Credit_Limit")
    Cust_ID = Node("Customer", name="Customer")
    Age = Node("Age", name= "Age")
    LastName = Node("LastName", name="LastName")
    Occupation = Node("Occupation", name="Occupation")
    email = Node("Email", name="Email")
    phone = Node("phone", name="phone")

    #Create Relationships
    print("Create Neo4j Nodes Relationships")
    Credit_Card_ID_has_Transaction_ID = Relationship(Credit_Card_ID, "INCLUDES",
Transaction_ID)
    Transaction_ID_has_Value = Relationship(Transaction_ID, "HAS_VALUE_OF",
Transaction_Value)
    Transaction_ID_isOn_Segment = Relationship(Transaction_ID, "IS_ON",
Transaction_Segment)
    Transaction_ID_has_Comments = Relationship(Transaction_ID,
"HAS_AS_COMMENTS", Comments)
    Credit_Card_has_Limit_of_Credit_Card_Limit = Relationship(Credit_Card_ID,
"HAS_LIMIT_OF",Credit_Limit)
    Cust_ID_has_Credit_Card_ID = Relationship(Cust_ID, "2,HAS", Credit_Card_ID)
    Cust_ID_has_Age = Relationship(Cust_ID, "IS_OF", Age)
    Cust_ID_hasName_LastName = Relationship(Cust_ID, "HAS_NAME_OF", LastName)
    Cust_ID_WorksAs_Occupation = Relationship(Cust_ID, "HAS_WORK_AS", Occupation)
    Cust_ID_hasEmail_Email = Relationship(Cust_ID, "HAS_EMAIL", email)
    Cust_ID_has_Phone = Relationship(Cust_ID, "HAS_PHONE", phone)

    #Create Graph
    graph.create(Credit_Card_ID_has_Transaction_ID)
    graph.create(Transaction_ID_has_Value)
    graph.create(Transaction_ID_isOn_Segment)
```

```
graph.create(Transaction_ID_has_Comments)
graph.create(Credit_Card_has_Limit_of_Credit_Card_Limit)
graph.create(Cust_ID_has_Credit_Card_ID)
graph.create(Cust_ID_has_Age)
graph.create(Cust_ID_hasName_LastName)
graph.create(Cust_ID_WorksAs_Occupation)
graph.create(Cust_ID_hasEmail_Email)
graph.create(Cust_ID_has_Phone)
```

Table 10: Connect to Neo4j function

Connect_To_Redis.py

```
#Libraries
import redis
from py2neo import authenticate,Graph
from ReadFromFilesFunctions import readFromNeo4jInput
from py2neo.packages.httpstream import http
http.socket_timeout = 9999

#Redis connection configuration
redis_host = "localhost"
redis_port = 6379
redis_password = ""
r = redis.StrictRedis(host="localhost", port=redis_port, password=redis_password, decode_responses=True)

#Function to Create KV Lists in Redis
def create_KV_Lists(result, nodeA, nodeB):
  # Loop through rows of dataframe by index i.e. from 0 to number of rows
    for i in range(0, result.shape[0]):
      rowSeries = result.iloc[i]
      #r-push to add elements in redis List
      r.rpush(nodeA+"_"+nodeB+"_"+str(rowSeries.values[0])+":",str(rowSeries.values[1]))
      print(nodeA+"_"+nodeB+"_"+str(rowSeries.values[0])+":")
      print(r.lrange(nodeA+"_"+nodeB+"_"+str(rowSeries.values[0])+":",0,-1))
```

Table 11: Connect to Redis function

Main.py

```
#Libraries
import sys
import pandas as pd
from py2neo import authenticate,Graph
from ReadFromFilesFunctions import readFromNeo4jInput
from Connect_To_Redis import create_KV_Lists
from ConnectToNeo4J  import createGraph
from py2neo.packages.httpstream import http
http.socket_timeout = 9999

#Function to call Neo4j and create key-value pairs in Redis
def callNeo4WithParameters(nodeA, nodeB):
    print("Connect To Neo4j")
    authenticate("localhost:7474", "neo4j", "123")
```

```
    graph = Graph("http://localhost:7474/db/data/")
    result= graph.cypher.execute("MATCH(a:"+nodeA+")-[r]-(b:"+nodeB+") RETURN r.P")
    result_df = pd.DataFrame(result.records, columns=result.columns)
    queryParameterNew = result_df['r.P'].values[0].split(';')
    result = readFromNeo4jInput(int(queryParameterNew[0]), queryParameterNew[1],
queryParameterNew[2], nodeA, nodeB)
    print("Create Redis lists")
    create_KV_Lists(result,nodeA,nodeB)

#Main Function --call from command line
#Parameters to call Main Function : nodeA, nodeB where nodeA, nodeB the nodes for key
value pairs
if __name__ == "__main__":
    nodeA = sys.argv[1]
    nodeB = sys.argv[2]
    callNeo4WithParameters(nodeA,nodeB)
```

Table 12: Main function

Combine_Keys.py

```
##Libraries
import sys
import pandas as pd
from py2neo import authenticate,Graph
import redis

redis_host = "localhost"
redis_port = 6379
redis_password = ""
r = redis.StrictRedis(host="localhost", port=redis_port, password=redis_password,
decode_responses=True)
authenticate("localhost:11013", "neo4j", "123")
graph = Graph("http://localhost:11013/db/data/")

##Function callNeo4jJsonParameters: In this function,
def callNeo4ForJsonParameters(node):
    authenticate("localhost:11013", "neo4j", "123")
    graph = Graph("http://localhost:11013/db/data/")
    result= graph.cypher.execute("MATCH (n:"+node+")-[r]-(b) return distinct b.name")
    result_df = pd.DataFrame(result.records, columns=result.columns)
    return(result_df)

#find keys based on root node
def findKeys(selected_nodes_df):
    namespace = globals()
    for i in range(0, selected_nodes_df.shape[0]):
        namespace['keys_id_%d'                                                    %
i]=callNeo4ForJsonParameters(selected_nodes_df['n.name'].values[i])
        if(len(selected_nodes_df)==1):
            keys_df = namespace['keys_id_%d' % i]
        else:
            if(i>0):
```

```
            d= i-1
            keys_df = namespace['keys_id_%d' % i].append(namespace['keys_id_%d' % d])
    return(keys_df['b.name'].values.tolist())



def findChildrenKeys(child_df,A,listOfKeys):
    childkeys = child_df['b.name'].values.tolist()
    childkeys.remove(A)
    finalchildren = []
    for ch in range(len(childkeys)):
        if(childkeys[ch] in listOfKeys):
            finalchildren.append(childkeys[ch])
    return(finalchildren)


def callRoot(A):
    ##Get from graph the nodes with Selected = 'Y'
    selected_nodes= graph.cypher.execute("MATCH(n) WHERE (n.Selected = 'Y') RETURN
n.name")
    selected_nodes_df                =                pd.DataFrame(selected_nodes.records,
columns=selected_nodes.columns)
    x = findKeys(selected_nodes_df)
    return(x)

def remove_Duplicates(x):
  return list(dict.fromkeys(x))
#Select root node


def combineKeys(listOfKeys):
    namespace = globals()
    K = []
    Keys = []
    for i in range(len(listOfKeys)):
        namespace['redis_keys_%d' % i]=r.keys('*'+A+"_"+listOfKeys[i]+'*')
        K.append(namespace['redis_keys_%d' % i])
        #Next step is to remove unused characters from list
        L= [x for x in K if x != []]
    for j in range(len(L)):
        for k in range(len(L[j])):
            key_name=L[j][k].split("_", 2)[2].replace(':', '').strip()
            Keys.append(key_name)
    return(Keys)

#k = listOfKeys[child], A= Customer, B = final_keys[key]
def retrieveList(A,B,k):
    namespace = globals()
    namespace['redis_values_%d'%child] = r.lrange(A+"_"+B+"_"+k+":",0,-1)
    return(namespace['redis_values_%d' % child])
```

```
def evaluate(k,A,B,listOfKeys,child):
    if(len(r.lrange(A+"_"+B+"_"+k+":",0,0))>0):
            L = retrieveList(A,B,k)
            #check if B has children
            childOfChild = callNeo4ForJsonParameters(B)
            rs = findChildrenKeys(childOfChild,A,listOfKeys)
            if(len(rs) ==0):
            #if B has no children
               #If child has only one value then print element
               if(len(L)==1):
                  f.write(B+":"+"""+L[0]+""")
                  if(child>1):
                     f.write(",\n")
                  else:
                     f.write("\n")
               else:
                  lst = ','.join(L)
                  f.write(B+":"+"["+lst+"]")
                  if(child>1):
                     f.write(",\n")
                  else:
                     f.write("\n")
            else:
               f.write(B+"_"+"s"+"["+"\n")
               for l in range(len(L)):
                  if(l>0):
                     f.write(",\n")
                  f.write("{\n")
                  f.write(B +":"+"""+L[l]+"""+","+"\n")
                  for cc in range(len(rs)):
                     evaluate(L[l],B,rs[cc],listOfKeys);
                  f.write("}"+"\n")
               f.write("]"+"\n")


####DEFINE root node. A is root, the node that user select as rootт node
if __name__ == "__main__":
    A = sys.argv[1]
    f= open("json_output.txt","w+")
    print("Start process...")
    listOfKeys=callRoot(A)
    if(A in listOfKeys):
       listOfKeys.remove(A)
    Key_Names = combineKeys(listOfKeys)
    ###REMOVE DUPLICATES FROM LIST KEY
    K = remove_Duplicates(Key_Names)
    print("Please Wait...")
####WRITE FILE TO json_output file##############

    for key in range(len(K)):
```

```
        if(key>0):
            f.write(",\n")
        f.write("{\n")
        f.write(A +":"+"'"+K[key]+"'"+",\n")
        for child in range(len(listOfKeys)):
            evaluate(K[key],A,listOfKeys[child],listOfKeys,child)
        #####call eval function
        f.write("}\n")
    f.close()
    print("End process...")
```

Table 13: Combine Keys function

Insert DB query

```
USE [PVDI_DB]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
        [Id] [int] IDENTITY(1,1) NOT NULL,
        [Customer] [nchar](7) NOT NULL,
        [FirstName] [nvarchar](50) NULL,
        [LastName] [nvarchar](250) NULL,
        [Gender] [nchar](10) NULL,
        [Age] [int] NULL,
        [Occupation] [nvarchar](250) NULL,
 CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED
(
        [Id] ASC,
        [Customer] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS =
ON) ON [PRIMARY]
) ON [PRIMARY]
GO
SET IDENTITY_INSERT [dbo].[Customer] ON

INSERT [dbo].[Customer] ([Id], [Customer], [FirstName], [LastName], [Gender], [Age],
[Occupation]) VALUES (1, N'CC67088', N'Person 1', N'Person 1', N'Female    ', 32, N'PMO')
INSERT [dbo].[Customer] ([Id], [Customer], [FirstName], [LastName], [Gender], [Age],
[Occupation]) VALUES (2, N'CC12076', N'Person 2', N'Person 2', N'Female    ', 29, N'BI
Developer')
INSERT [dbo].[Customer] ([Id], [Customer], [FirstName], [LastName], [Gender], [Age],
[Occupation]) VALUES (3, N'CC49168', N'Person 3 ', N'Person 3', N'Male      ', 33,
N'Analytics')
INSERT [dbo].[Customer] ([Id], [Customer], [FirstName], [LastName], [Gender], [Age],
[Occupation]) VALUES (4, N'CC66746', N'Person 4', N'Person 4', N'Female    ', 32, N'CIO')
INSERT [dbo].[Customer] ([Id], [Customer], [FirstName], [LastName], [Gender], [Age],
[Occupation]) VALUES (5, N'CC28930', N'Person 5', N'Person 5', N'Male      ', 30, N'CTO')
SET IDENTITY_INSERT [dbo].[Customer] OFF
```

Table 14: Insert DB query

Neo4J Properties

```
SET PROPERTIES :

##CUSTOMER DATA SET RELATIONSHIPS
MATCH (a {name:"Customer"})-[r]-(b {name:"Age"})
SET r.P = "0;sql;select-Customer-Age-from dbo.Customer"

MATCH (a {name:"Customer"})-[r]-(b {name:"LastName"})
SET r.P = "0;sql;select-Customer-LastName-from dbo.Customer"

MATCH (a {name:"Customer"})-[r]-(b {name:"Occupation"})
SET r.P = "0;sql;select-Customer-Occupation-from dbo.Customer"

MATCH (a {name:"Customer"})-[r]-(b {name:"Email"})
SET r.P = "3;json;Customer,Email"

MATCH (a {name:"Customer"})-[r]-(b {name:"phone"})
SET r.P = "3;json;Customer,phone"

###ADD NODE WITH PROPERTY
MATCH (n:Customer)
SET n.Selected = 'Y'
RETURN n

MATCH (n:CreditCard)
SET n.Selected = 'Y'
RETURN n


MATCH (n:Transaction)
SET n.Selected = 'N'
RETURN n

##END CUSTOMER DATA RELATIONSHIPS


## CREDIT CARD ID RELATIONSHIPS
MATCH (a {name:"Customer"})-[r]-(b {name:"CreditCard"})
SET r.P = "2;excel;CreditCard,Customer"

MATCH (a {name:"CreditCard"})-[r]-(b {name:"Transaction"})
SET r.P = "1;csv;Transaction,CreditCard"

MATCH (a {name:"CreditCard"})-[r]-(b {name:"Credit_Limit"})
SET r.P = "2;excel;CreditCard,Credit_Limit"

#END CREDIT CARD ID RELATIONSHIPS

#TRANSACTIONS RELATIONSHIPS
```

```
MATCH (a {name:"Transaction"})-[r]-(b {name:"Transaction_Value"})
SET r.P = "4;csv;Transaction,Transaction_Value"

MATCH (a {name:"Transaction"})-[r]-(b {name:"Transaction_Segment"})
SET r.P = "4;csv;Transaction,Transaction_Segment"

MATCH (a {name:"Transaction"})-[r]-(b {name:"comments"})
SET r.P = "6;json;Transaction,comments"

#END TRANSACTIONS RELATIONSHIPS

#DELETE ALL COMMAND
MATCH (n)
DETACH DELETE n
```

Table 15: Neo4j Properties

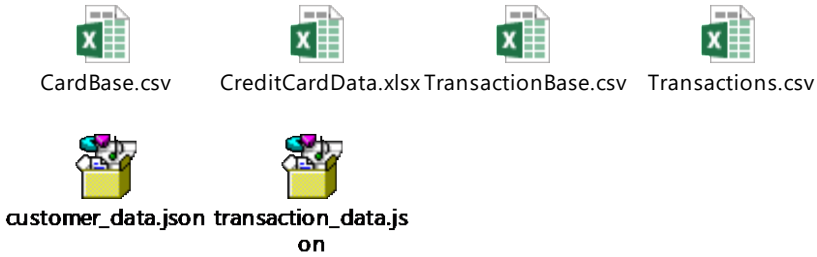
Config.json

```
{
  "datasource": [
    {
      "ID": "0",
      "type": "sql",
      "Connection": "DRIVER={SQL Server Native Client 11.0};SERVER=LAPTOP-L9MT57GS;DATABASE=Thesis;Trusted_Connection=yes",
            "output":"C:/outputDB.csv"
    },
    {
      "ID": "1",
      "type": "csv",
      "Connection": "C:/Transactions.csv",
            "output":"C:/outputCsv.csv"
    },
    {
      "ID": "2",
      "type": "excel",
      "Connection": "C:/CreditCardData.xlsx",
            "output":"C:/outputExcel.csv"
    },
    {
      "ID": "3",
      "type": "json",
      "Connection": "C:/customer_data.json",
            "output":"C:/outputJson.csv"
    },
          {
      "ID": "4",
      "type": "csv",
      "Connection": "C:/TransactionBase.csv",
            "output":"C:/outputcsv11.csv"
    }
        ,
```

```
        {
    "ID": "5",
    "type": "csv",
    "Connection": "C:/CardBase.csv",
          "output":"C:/outputcsv111.csv"
  }
        ,
        {
    "ID": "6",
    "type": "json",
    "Connection": "C:/transaction_data.json",
          "output":"C:/outputJson.csv"
  }
 ]
}
```

Table 16: configuration file

Test Data



****** End of Document******