A Conceptual Approach to Traffic Data Wrangling

DOI:
10.1007/978-3-319-60795-5_2

Document Version
Accepted author manuscript

A Conceptual Approach to Traffic Data Wrangling

Mashael Aljubairah[1], Sandra Sampaio[2], Hapsoro Adi Permana[3], and Pedro Sampaio[4]

[1] School of Computer Science, University of Manchester
mashael.al-jubairah@postgrad.manchester.ac.uk
[2] School of Computer Science, University of Manchester
s.sampaio@mancheste r.ac.uk
[3] School of Computer Science, University of Manchester
hapsoroadi.permana@postgrad.manchester.ac.uk
[4] Alliance Manchester Business School, University of Manchester
p.sampaio@manchester.ac.uk

Abstract. Data Wrangling (DW) is the subject of growing interest given its potential to improve data quality. DW applies interactive and iterative data profiling, cleaning, transformation, integration and visualization operations to improve the quality of data. Several domain independent DW tools have been developed to tackle data quality issues across domains. Using generic data wrangling tools requires a time-consuming and costly DW process often involving advanced IT knowledge beyond the skills set of traffic analysts. In this paper, we propose a conceptual approach to data wrangling for traffic data by creating a domain-specific language for specifying traffic data wrangling tasks and an abstract set of wrangling operators that serve as the target conceptual construct for mapping domain-specific wrangling tasks. The conceptual approach discussed in this paper is tool-independent and platform agnostic and can be mapped into specific implementations of DW functions available in existing scripting languages and tools such as R, Python, Trifacta. Our aim is to enable a typical traffic analyst without expert Data Science knowledge to be able to perform basic DW tasks relevant to his domain.

Keywords: Data Wrangling, Data Transformation and Quality, Conceptual Wrangling Approaches

1 Introduction

Decision makers in different domains, such as healthcare, education and transportation, can gain significant advantages from the enormous volume of available data obtained from various data collection methods, such as site-based sensors, cell-phone tracking [1] and social media. However, data collected via these methods are prone to data quality problems, such as inaccuracy, incompleteness and

heterogeneity [2] [3]. More recently, data management techniques for data profiling, cleaning and integration have been adapted to improve the quality of large amounts of raw data, in preparation for analysis. The combination of these data management tasks is often called Data Wrangling (DW), generally defined as "the process by which the data required by an application is identified, extracted, cleaned and integrated, to yield a data set that is suitable for exploration and analysis" [4]. According to IBM [5], data analysts spend around 70% of their time conducting DW activities. Being an interactive and iterative process that involves the application of a variety of data management methods, and that generally lacks a rigid methodology across application domains, DW is often regarded as a highly complex job requiring advanced skills and domain expertise.

Data analysts typically perform DW tasks by using one or a combination of the two following approaches: (i) by programming their own DW applications, using languages such as Python, Java and R; and (ii) by interacting with existing DW tools, which often provide access via a Graphical User Interface (GUI). While approaches (i) and (ii) provide benefits, they also have shortcomings. Approach (i), for example, is often associated with completeness of functionality for fulfilling the requirements of the application in consideration; however, it also involves complex application development, advanced programming skills and brittle solutions that cannot be easily applied over data from other sources than the ones for which the solution was originally designed. On the other hand, approach (ii) is often associated with ease of user interaction, limited need for programming skills, provision of generic functionality that cannot be easily adapted to fulfil specific functional requirements, need for use of multiple tools to perform a single DW job, and limited opportunity for optimization.

To mitigate the limitations identified in approaches (i) and (ii), we propose a conceptual approach and an architectural solution for DW that combines advantages from (i) and (ii), while offering user interaction via a GUI and a high-level and domain-specific declarative language for simple DW tasks. The proposed architecture combines functionality from multiple DW tools and access to multiple data sources, by using Web Services technology. The result is an extensible DW tool, able to take advantage of DW functionality implemented within a variety of existing DW tools, and that provides extensibility and flexibility to allow data analysts to add functionality specific to the requirements of the application in consideration, creating a rich set of DW functions, that can be combined to accomplish simple and complex, general and domain specific DW tasks. For that, high-level user DW requests are automatically mapped into a set of conceptual DW constructs, that are ultimately translated into an execution plan represented as a workflow combining local as well as remote DW functions implemented across a multitude of tools. The proposed approach is tested with use cases from the Urban Traffic Domain, in which DW tasks associated with common data analysis requests by traffic analysts are executed over an implementation of the approach. The paper is organized as follows: Section 2 provides

a literature review. Section 3 provides the conceptual DW approach including the architecture, conceptual and physical layer design sections, and discussion of implementation aspects. Section 4 provides the conclusions and future work.

2 Literature Review

There is a body of research to facilitate DW via Graphical User Interfaces (GUIs) and Domain-Specific Languages (DSLs). For example, in the work of *Kandel et al.* [6], the complexity of conducting DW was decreased, by associating DW functionality with data visualization constructs, allowing users to conduct DW via a visual interface. Tools offering DW functionality, such as Trifacta [7] and OpenRefine [8], provide concise DSLs combined with GUIs to isolate users from the complexities involved in the wrangling process. Even though both the Trifacta Wrangling Language and the General Refine Expression Language provide data transformation capabilities in Trifacta and OpenRefine, respectively, these languages are not high-level and declarative, and force users to use low-level language constructs. In addition, the level of completeness of functionality provided by these tools varies based on the DW requirements associated with the task at hand, as well as the characteristics of the target data. Considered in isolation, each of these tools will often fail to provide all the operations needed to effectively support the DW capabilities that complex information management problems typically require. A functionality-based comparison for the tools is provided in Tables 3 and 4 in Appendix A.

The difficulties in finding a tool that offers all the functionality required to perform DW tasks forces data analysts to face a steep learning curve before familiarising themselves with multiple tools and experiencing a rather laborious and complex process, in which data often needs to be transformed/reformatted to be transferred between different tools. In addition, data analysts may still have to use low-level programming constructs implemented in languages such as R, Python or Java to be able to customise code and solve specific data quality issues, despite using the DW tools. For example, Bluetooth-based road sensors used to collect traffic data often produce duplicate records for the same moving object due to multiple passengers carrying switched-on Bluetooth devices in a vehicle. For removing duplicates, in this case, multiple attributes need to be considered, such as vehicle identifier, time and location of detection, and device MAC address. However, tools offering DW functionality such as Trifacta Wrangler, OpenRefine and Talend data preparation only provide generic functionality for removing duplicates and so, cannot easily identify non-identical records as duplicates. In addition, because road sensors are generally prone to failure due to environmental conditions and are often able to detect only moving objects equipped with a switched on Bluetooth device, missing data is a common problem in traffic data sets. In the traffic domain, missing data can be replaced with data from the nearest periods or from similar locations or both [9]. Therefore, `Spatial Joins` using latitude and longitude information are often required

to address missing data problems and are often not supported by general DW tools. Outliers are also difficult to address in generic DW tools due to the need to include semantic-based information to distingush between outliers and noise. Although there are outlier detection operations distributed across several DW tools, they do not correlate data with other attributes that are important to decide whether the value is an outlier.

3 A Conceptual Approach to Traffic Data Wrangling

3.1 Architectural Overview

Fig. 1 illustrates the proposed Data Wrangling (DW) architecture. DW requests are expressed in the Declarative Data Wrangling Language (D^2WL) and submitted for parsing, during which validation of a D^2WL expression is carried out by checking the relevant data sources and other schema information, using meta data. At the end of parsing, a number of expressions in the Data Wrangling Language (DWL) is generated and submitted to optimization. In the current prototype, optimization is based on static information and heuristics, but future work will focus on adaptive and dynamic optimization, based on a cost model. The declarative language is designed for data analysts with limited or no programming skills and so it is based on a small number of clauses that define the location and format of input data sources (the `FROM` clause), the location and format of results (the `TO` clause), the main data wrangling activity to be carried out (the `WRANGLE` clause), and other data wrangling activities using clauses such as `GROUPBY`. A number of simple functions is also defined within D^2WL to facilitate the expression of specific data wrangling functions, such as filling in of missing values using function `MISSINGDATA(listofattributes)` and `BY aggregate` operation, which defines which data attributes should have their missing values filled in, and how. Fig. 3 provides an example D^2WL expression that requests attributes `gap` and `headway` of a comma-separated values (CSV) file to be filled in by the average value for each of these attributes. Fig. 2 provides the same DW request expressed in English. Note that gap (the time distance between the rear bumper of one vehicle and the front bumper of the vehicle behind it) and headway (the time distance between the front bumpers of two subsequent vehicles) are important traffic congestion indicators, because there is a direct correlation between gap or headway between two vehicles and vehicle speed, and an inverse correlation between the gap or headway and traffic volume. The design of the language is based on an analysis of the most common DW requests by traffic analysts and is able to express a majority of traffic DW requests. The architecture is designed to minimize human interaction by providing automatic translation of D^2WL expressions into DWL expressions, which are composed of a series of conceptual DW operators needed to perform the requested DW tasks. A GUI will also be provided, via which analysts are able to build complex DW requests, by connecting data sources and abstractions of DW operations as a workflow. Ultimately, each optimized DWL expression is mapped into a number

of physical DW operations available as services and possibly from different tools and sources.

3.2 The Conceptual Layer

The model of data and computation of the proposed Data Wrangling Language (DWL) is functional, with a number of data types and functions, and was designed to be extensible, allowing easy incorporation of additional functions. The data types supported are classified into two main types: atomic and aggregate. Atomic data types include string, character, date, time, boolean and numeric data, such as integer and float. While aggregate data types include general collection types, such as bag, representing an unordered collection of objects allowing duplicates, list, an ordered collection of objects allowing duplicates, set, an unordered collection of objects with no duplicates, and tuple, a record containing objects with different data types. The following symbols are relevant to the DWL example described in Fig. 4: The symbol ⌈⌉ is used to represent an empty bag, <<>> is used to denote an empty list, [] represents an empty set and {} is used to represent an empty tuple. The Collection data type is a generalization of all four collection types, and its constructor is represented by (). An additional data type is Graph, which models visual representations of data.

Fig. 4 describes a DWL expression for the D^2WL expression in Fig. 3. The `read` operator takes an `URL` as input identifying the name, format and location of the input file, possibly on a remote server, and brings the file into context. In this example, this CSV file is described in Appendix A. The `enrichTimeStamp` operator takes the `columnname` of a date attribute, and formats it according to the provided `format` and `startdate` to add date, month and year information to the attribute. The `addColumn` operator takes a `listofexpressions` containing pairs of the type `<<columnname, expression>>`, describing the names of new columns to be inserted into the file and expressions associated with the column names defining how each new column is to be populated. In this example, the pairs are as follows: `<<hour, hour=extractHour(completetimestamp)>, <dayofweek, dayofweek=extractDayofweek(completetimestamp)>>`. `extractHour` is used to extract the hour from `completetimestamp` which is the attribute generated by `enrichTimeStamp`. `extractDayofweek` is similar to `extractHour` but used to extract the weekday from a date. The result of operator `addColumn` is the input to `selectColumn` with a `listofcolumnnames`, representing the attributes to be projected from the file. The result of `selectColumn` is input to `groupBy`, which is also provided with a `listofcolumnames` that represent the attributes by which records are to be grouped. The `fill` operator then takes the result of `groupBy` and replaces the zero values in columns gap and headway with the value resulting from the evaluation of the expression provided in `listofexpressions`. In the example, the expressions in the list are `<< gap=average(gap), headway=average(headway)>>`, based on the `listofconditions` which are `<<if gap==0, if headway==0>>`. Note that the aggregate operations defined in the list of expressions are impacted by how

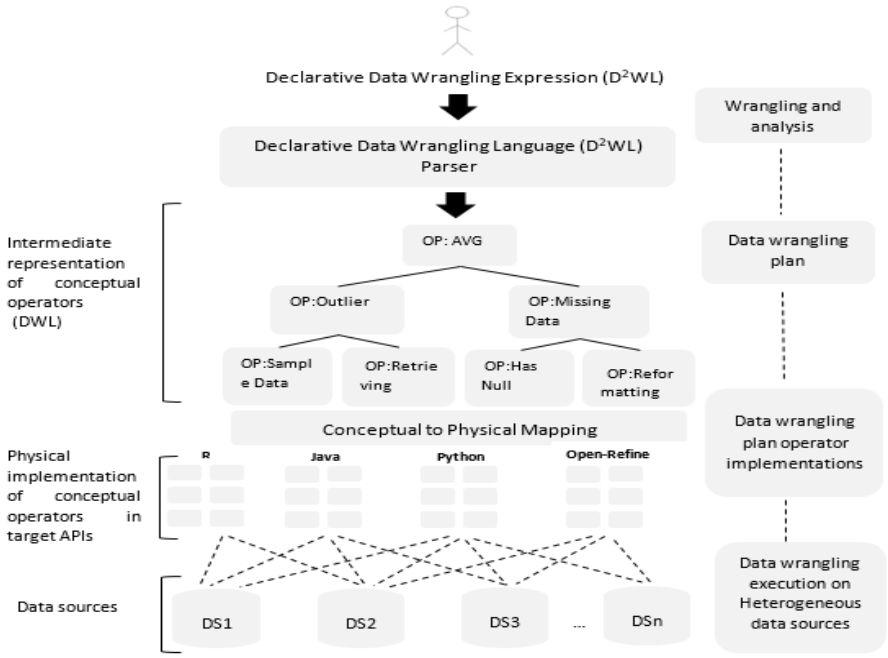

Fig. 1: Data wrangling system architecture.

"Fill in missing values using the average value for headway and gap, and based on observation site, vehicle direction, lane, hour, and day of week, the values for headway and gap could be inferred"

Fig. 2: English expression for the example DW task.

```
FROM  http://www.informationfortraffic.com/Inductive_loops.csv
TO  http://www.informationfortraffic.com/Inductive  loops.csv
WRANGLE  MISSINGDATA(gap, headway)
BY  Average
GROUPBY  (siteID,vehicleDirection,lane,hour,DayOfWeek)
```

Fig. 3: D^2WL expression for the example DW task.

$$
\begin{aligned}
&Result \Leftarrow view \\
&(fill_{<<listofexpression>>,<<listofcondition>>} \\
&(groupBy_{<<listofcolumnname>>} \\
&(selectColumn_{<<listofcolumnname>>} \\
&(addColumn_{<<listofexpression>>} \\
&(enrichTimeStamp_{columnname,format,startdate} \\
&(read(URL)))))))
\end{aligned}
$$

Fig. 4: Conceptual DWL expression for the example DW task.

records are grouped by a `groupBy` operator. Finally, the `view` operator allows the user to visualize the result on the screen. A further description of each operation is provided in Appendix B.

3.3 The Physical Layer

The Taverna [10] workflow management system is currently being used to call and manage several DW operations implemented within different target tools. The rationale behind the choice of Taverna in the current implementation of the proposed system is its ability to manage complex workflows that require connections to remote services, as well as its ability to allow the annotation and storage of a potentially large number of workflows. Fig. 5 shows an example of how a conceptual DW operation such as `read` (described in Table 5 of Appendix B) is mapped into a low-level Taverna workflow. Each DW operation is represented as a Taverna workflow, so that the conceptual DW operation name is mapped to a workflow name, the operation argument(s) are mapped into the workflow input port(s) and the operation result(s) are mapped into the workflow output port(s). Each Taverna workflow of a mapped conceptual DW operation includes three processors each of which is associated with a Beanshell or REST service to specify how its functionality is achieved. These services and processors process and pass the input to the workflow via data links which have sink and source variables to identify the connection direction and obtain the target result. Each input and output to a processor is associated with the required input and output to its task-related activity, and the corresponding data link is used to associate the output of the processor with the input to the next processor. Fig. 6 shows the actual execution steps of the Taverna workflow representing the `read` operation which is implemented as an R web service. The Beanshell service, `EncodeURLfile`, is responsible for encoding the input to the REST service, `read.csv`, implemented in an R server. This is followed by the Beanshell service, `getSessionKey`, which obtains the session key associated with the data set brought into context by the `read.csv` function and makes it available to the next DW conceptual operator, another workflow that will process the data set.

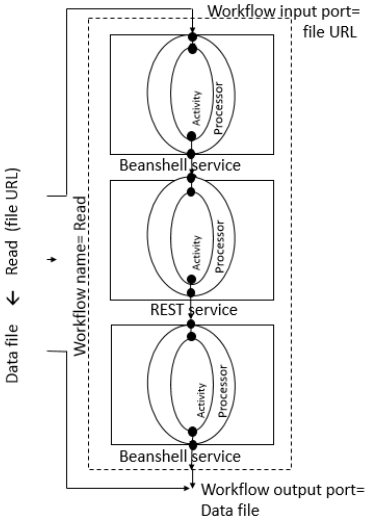

Fig. 5: Details of the Taverna workflow for the conceptual read operation.

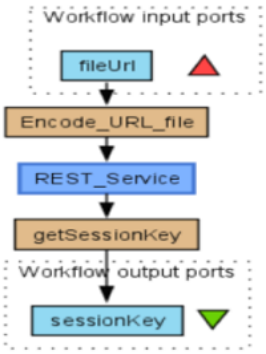

Fig. 6: Execution steps of a Taverna workflow representing the conceptual read operation.

In Section 3.4, further implementation details for the example DW task in Fig. 4 are provided.

3.4 Implementation of DW operations in Taverna

Fig. 7 shows the realization of the DW task shown in Fig. 4 as a composition of Taverna workflows. Note that the DWL operators in Fig.4 have been mapped into one or more low-level functions available from two remote servers, one being an R server, and the other one offering DW functionality via simple Python scripts. As described in Section 3.3, Beanshell services have been used to facilitate the HTTP-based communication of data between the various workflows, such as the URL of the file to be input to the task, and a session key associated with an intermediate result that is passed from one workflow to the next. Therefore, each workflow is, in turn, composed of three smaller services, specifically one REST service that encapsulates the main data processing activity associated with the workflow, and two enveloping Beanshell services, the first serving as an input data facilitator and the other one dedicated to extracting the data session key to be output to the next workflow, as Fig. 5 suggests.

In the workflow composition in Fig. 7 the DWL operator, `fill`, has been mapped into the following calls to R functions: `mutate`, to create new columns and impute missing values for `gap` and `headway` using expressions and column names as parameters. The expressions include two additional R operators, `replace` and `average`, to replace the missing values using the average value. Two Beanshell services, `imputeexpression` and `colnamesimpute` were used to

provide the `mutate` parameters values; the `groupBy` operator has been mapped into R's `groupby` operator to group the data set based on the provided attributes given by a Beanshell service named `colnamesgroupby`. Similarly, the `selectColumn` operator has been mapped into R's `selectcolumns` to select the required columns provided by `colnamesslect` Beanshell service; the `addColumn` operator has been mapped into R's `mutate` function to create new columns contain information of hour and day of week based on the expressions and column names parameters provided by Beanshell services `expressionmutate2` and `colnamesmutate2`, respectively. The `read` operator has been mapped into R `read.csv` function to read a remote CSV file, and the `enrichTimeStamp` has been mapped into `completetimestamp` Python function to convert the time format from <Minute><Second><Milliseconds> to a more readable format (i.e. 24-hour format) and calculate the date. This operation required `begindate`, `datecolumnname` and `trafficdataurl` as parameters.These parameters are also the workflow input ports as shown in Fig.7. In addition, Taverna allows downloading of the file using the REST service component named `getcsv`, and the data set was stored into a local CSV file using the Taverna input-output component named `writetextfile`. The different colours used in Fig.7 indicate different services provided by Taverna. The pink rectangles indicate the encapsulated individual workflows, the brown rectangles indicate Beanshell services, the blue rectangles indicate REST service and the purple rectangle indicates Taverna input-output component.

4 Conclusions and Future Work

Due to the complexity associated with the DW process, there is often the need to apply several DW tools, presenting significant challenges to the data analyst. To address these challenges, a novel architectural approach to DW has been proposed. The proposed architecture for DW combines advantages from the existing DW approaches by providing abstract and domain-specific DW constructs without the need for end users to learn low-level programming APIs. In addition, this approach allows data analysts to take advantage of the functionalities available in existing tools. In addition to the Declarative Data Wrangling Language, D^2WL, the conceptual Data Wrangling Language (DWL) can also be complemented by other high-level and declarative domain-specific languages relating to other domains such as education, healthcare and finance. We are currently developing a graphical user interface for traffic analysts to specify D^2WL expressions and also assistive technology to map from high-level requests to optimised DW strategies implemented in the target platforms. To evaluate the proposed tool, a number of end user experiments will be conducted where analysts from traffic domain are to conduct data wrangling using the tool. A detailed evaluation of the tool will be performed based on both user productivity and tool performance. The user productivity will be measured based on criteria such as learning curve, ease of use, functionality coverage and level of user interaction/effort by quantifying the number of steps required to perform a number of DW tasks of varying complexity. To collect the results, a questionnaire will be provided to data wranglers

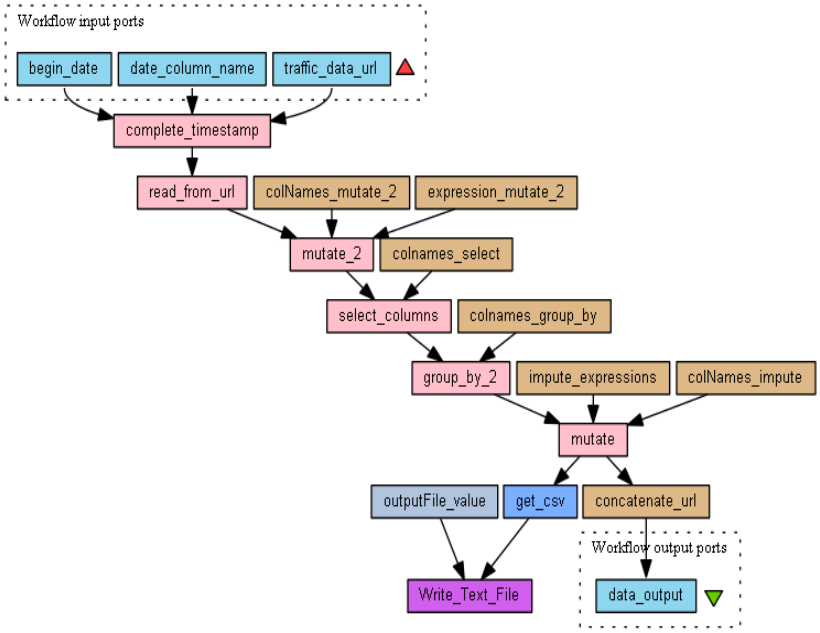

Fig. 7: Implementation of DW task using Taverna.

(i.e. end-users) and performance of existing DW tools will be compared to results obtained from our experiments.

References

[1] J Lopes, J Bento, E Huang, C Antoniou, and Moshe Ben-Akiva. Traffic and mobility data collection for real-time applications. In *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*, pages 216–223. IEEE, 2010.

[2] Jon Hutchins, Alexander Ihler, and Padhraic Smyth. Probabilistic analysis of a large-scale urban traffic sensor data set. In *Knowledge Discovery from Sensor Data*, pages 94–114. Springer, 2010.

[3] HV Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[4] Tim Furche, Georg Gottlob, Leonid Libkin, Giorgio Orsi, and Norman W Paton. Data wrangling for big data: Challenges and opportunities. In *EDBT*, pages 473–478, 2016.

[5] Ignacio Terrizzano, Peter M Schwarz, Mary Roth, and John E Colino. Data wrangling: The challenging yourney from the wild to the lake. In *CIDR*, 2015.

[6] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and

Paolo Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization*, 10(4):271–288, 2011.

[7] Trifacta Wrangler Enterprise. Trifacta wrangler. https://www.trifacta.com/, 2015. [Online; accessed 05-01-2016].

[8] open source community Google. Openrefine tool. https://github.com/OpenRefine/OpenRefine/wiki/General-Refine-Expression-Language, 2015. [Online; accessed 28-12-2015].

[9] Chenjuan Guo, Christian S Jensen, and Bin Yang. Towards total traffic awareness. *ACM SIGMOD Record*, 43(3):18–23, 2014.

[10] Apache Software Foundation. Taverna. https://taverna.incubator.apache.org/[, 2015. [Online; accessed 30-01-2017].

[11] Transport for Great Manchester. Transport for great manchester. http://www.tfgm.com/Pages/default.aspx, 2015. [Online; accessed 17-06-2015].

A Comparison of three widely used data wrangling tools

This Appendix illustrates the capabilities and limitations of three widely used data wrangling tools in fulfilling a data wrangling task using the traffic data sets depicted in Tables 1 and 2, and illuestrated in Table 3. In addition, it provides a comparion of the three data wrangling tools according to the import and export data format as shown in Table 4. The data wrangling task described as " Retrieve the value of the average speed of vehicles passing road R, (e.g. A572 Manchester Road, Astley Green), on a day of week D (e.g. Monday) at a time interval T (e.g. 8:00 to 9:00)."

B Description of data wrangling operations

This Appendix provides a description of data wrangling operations as shown in Tables 5,6,7,8,9,10 and 11.

Table 1: Data collected by inductive loops, taken from TfGM database [11]

Site ID	Date	Lane	Lane Name	Direction	Direction Name	Class Scheme	Class	Class Name	Length	Headway	Gap	Speed (mph)	Weight	Vehicle Id	Flags	Flag Text	Num Axles
'000000001304	00:00:29	1	EB	1	East	3	2	Car	14.1			33.6			0		0
'000000001304	00:01:30	1	EB	1	East	3	2	Car	14.4	60.8	60.5	30.4			0		0

Table 2: Static data, taken from TfGM database [11]

Site ID	Site Name	Description	Speed Limit (mph)	Grid		Orientation	Longitude	Latitude	Bearing	Parameters
1304	1304	A572 Manchester Road, Astley Green	30	369151000000	E		-2.46646	53.49416	90	noexport=1
1305	1305	A572 Leigh Road, Boothstown	40	373097000000	w		-2.40707	53.50274	270	noexport=1

Table 3: Applying the data wrangling task on data wrangling tools

Stage No.	Description	OpenRefine	Trifacta -free desktop version	Talend-data preparation tool
Stage 1	Check the datasets characteristics:big data	Not supported.	Supported.	Not supported.
Stage 2	Check the datasets characteristics:CSV data format	Supported.	Supported.	Supported.
Stage 3	Enrich the dataset by Time and Date.	Not supported.	Not supported.	Not supported.
Stage 4	Extract the last four digits of Site ID.	Supported.	Supported.	Supported.
Stage 5	Extract weekday.	Supported.	Supported.	Supported.
Stage 6	Join the data with static data using SiteID as a key.	Supported.	Supported.	Supported.
Stage 7	Filter the data based on weekday, road name and time.	Supported.	Supported.	Supported.
Stage 8	Calculate the average.	Not supported.	Supported.	Not supported.

Table 4: Data format-based comparison of the data wrangling tools

comparison criteria	Trifacta -free desktop version	OpenRefine	Talend-data preparation tool
Input data format			
Comma Separated Values (CSV)	Supported.	Supported.	Supported.
JavaScript Object Notation(JSON)	Not supported.	Supported.	Not supported.
Extensible Markup Language (XML)	Not supported.	Supported.	Not supported.
Export data format			
Comma Separated Values (CSV)	Supported.	Supported.	Supported.
JavaScript Object Notation(JSON)	Not supported.	Not supported.	Not supported.
Extensible Markup Language (XML)	Not supported.	Not supported.	Not supported.

Table 5: Operation 1

Operation name	read
Format/ Operation expression	read(String)
Arguments/ Inputs	String: the string includes the file name, type and its location on a remote server which can be represented in a URL.
Description	"read" reads a file in a remote server by specifying the name, type and its location using a URL.
Output	Collection data type.

Table 6: Operation 2

Operation name	enrichTimeStamp
Format/ Operation expression	enrichTimeStamp (({Literal}), String, Time, Date)
Arguments/ Inputs	({Literal}): the data set where you want to perform the "enrichTimeStamp" operation. The collection can be a bag, set or list of tuple data type. String: the column name in your data set which includes Time data that need to be converted. Time: the specified Time format provided by a user. Date: the start date specified by a user.
Description	"enrichTimeStamp" enriches a data set by converting the time format from MM:SS:MS to time HH:MM:SS and infer date as DD/MM/YYYY . This would be based on the provided Time format and start date.
Output	Collection data type.

Table 7: Operation 3

Operation name	addColumn
Format/ Operation expression	addColumn (({Literal}), << Expression >>)
Arguments/ Inputs	({Literal}): the data set where you want to perform the "addColumn" operation. The collection can be a bag, set or list of tuple data type. << Expression >>: list of expressions contains the name/s of new column/s and how the data will be filled in the new column/s. The expression can be: (1) comparison expressions: results in a value of either TRUE or FALSE. The expression can include relational operators and operators such as AND, OR, XOR, NOR, and NOT. (2) Arithmetic expression: results in a numeric value. The expression can include arithmetic operators (e.g. +, -, *, /, %). The expression can also include one or more operations.
Description	"addColumn" enriches the data set by adding new column/s filled based on the specification written in the expression.
Output	Collection data type.

Table 8: Operation 4

Operation name	selectColumn
Format/ Operation expression	selectColumn (({Literal}), << String >>)
Arguments/ Inputs	({Literal}): the data set where you want to perform the "selectColumn" operation. The collection can be a bag, set or list of tuple data type.<<String>>: list of column names represents the selected columns.
Description	"selectColumn" selects specific column/s from a data set.
Output	Collection data type.

Table 9: Operation 5

Operation name	groupBy
Format/ Operation expression	groupBy (({Literal}), << String >>)
Arguments/ Inputs	({Literal}): the data set where you want to perform the "groupBy" operation. The collection can be a bag, set or list of tuple data type. << String >>: list of column name/s.
Description	"groupBy" arranges identical data into groups based on the specified columns.
Output	Collection data type.

Table 10: Operation 6

Operation name	fill
Format/ Operation expression	fill ({Literal}), << Expression >>, << Expression >>)
Arguments/ Inputs	({Literal}): the data set where you want to perform the "fill" operation. The collection can be a bag, set or list of tuple data type. << Expression >>: list of expressions contains the name/s of columns which have missing values and and how the values will be resolved./s. The expression can also include one or more operations. << Expression >>: list of conditional expressions contains the name/s of columns which have missing values and conditions on the values of these columns to identify when the value is considered as missing and should be replaced. The expression is comparison expressions which result in a value of either TRUE or FALSE. The expression can include relational operators and operators such as AND, OR, XOR, NOR, and NOT.
Description	"fill" replaces every row in specified column/s by the given value based on the conditional expression provided by a user.
Output	Collection data type.

Table 11: Operation 7

Operation name	view
Format/ Operation expression	view (({Literal}))
Arguments/ Inputs	({Literal}): the data set where you want to perform the "view" operation. The collection can be a bag, set or list of tuple data type.
Description	"view" allows users to visualize a file on a screen.