9

Overview – The Social Media Data Processing Pipeline

David M. Brown, Adriana Soto-Corominas, Juan Luis Suárez and Javier de la Rosa

This chapter provides a broad introduction to the modelling, cleaning, and transformation techniques that must be applied to social media data before it can be imported into storage and analysis software. While each of the above topics in itself encompasses a wide range of issues, they are also inextricably related in that each relies in some way upon the others. In order to discuss these processes as a group, we employ the term data processing to describe the preparatory phase between data collection and data analysis. The sections that follow demonstrate how data processing can be broken down into a pipeline of three phases:

- In the first phase, **modelling**, the data is manually evaluated for structure and meaning by identifying entities, their attributes, and how they are related. This information is then mapped to a data model, a schematic that will determine the requirements for cleaning and transformation. Also, the data model is often translated to a database schema in order to prepare for data import into a database management system.
- In the next phase, **cleaning**, the data is analyzed for possible sources of inconsistencies that could interfere with analysis. Inconsistent entries are then either removed, or resolved using one of a variety of statistical techniques. Furthermore, improperly formatted fields can be managed during the cleaning phase.
- Finally, in the **transformation** stage, the data is read from a data source using either programmatic techniques, or software designed for data manipulation. It is then parsed in order to extract and structure the information required by the data model. Finally, the data is output in a format that is compatible with the import system of the chosen storage or analysis software.

Each of these phases will be presented as a separate section that provides an overview of relevant concepts, as well as examples that put them into practice using state of the art tools and techniques. Due to the typically linked nature of social media data – think Twitter, Facebook, LinkedIn – this chapter focuses on preparing data for social network

style analysis, which seeks to understand social structure and behavior by modelling reality as a collection of nodes (things) connected to one another through their interactions, or relationships (McCulloh, Armstrong, and Johnson, 2013). In the context of social media, network analysis seeks to understand how individuals interact within loosely connected information networks through the exchange of digital artifacts, such as Facebook posts and tweets on Twitter (Rainie and Wellman, 2012). All of the provided examples are based on Twitter, and were tested using a subset of the Paris Attacks Twitter Dataset, which consists of approximately 40,000 tweets formatted as JSON records collected by the CulturePlex Laboratory in a twenty-four hour period following the terrorist attacks in Paris, France on November 13, 2015. The subset was selected to include only geolocated tweets, which contain latitude/longitude information that identifies the geographic origin of the tweet (Sloan and Morgan, 2015).

SOCIAL MEDIA DATA MODELLING – FROM DOMAIN TO DATABASE

Data modelling is a broad topic that encompasses a variety of techniques and concepts. Generally speaking, data modelling attempts to recognize and define the structure and meaning contained within data in order to create a model, which is often represented as a schematic diagram. In essence, a data model is a calculated abstraction of a real world domain that is specifically designed to meet certain storage and analysis needs (Elmasri and Navathe, 2010; Robinson, Webber, and Eifrem, 2015). It guides the rest of the data processing procedures in that it determines the structure of the required output, and is typically mapped to a database system, thus defining how data will be stored and accessed. Traditionally, data is modelled at three levels:

- The *conceptual model*, sometimes referred to as a whiteboard model, describes the semantics of the data set. It provides a conceptual map that highlights the main entity types present in the data and how they are related to one another. This model will also be referred to as a domain model, because its goal is to represent the sphere of activity or knowledge associated with the data.
- The *logical model* is based on the conceptual model, but is created in anticipation of mapping the model to a database system. This model takes into account concepts such as access paths in order to increase efficiency in data storage, retrieval, and analysis; however, the logical model is generally considered to be independent of any particular technology.
- The *physical model* determines how data will physically be stored on a computer relative to the specifics of the database system being used. This model takes into account internal storage structures, such as data types, storage space requirements, indexing, etc.

While each of these levels is important for a complete discussion of data modelling, both logical and physical modelling can become very complex topics, especially when the modelling process targets relational databases. In light of this complexity, this chapter focuses primarily on the property graph model utilized by graph databases, which are designed and optimized to work with highly connected data. Furthermore, property graph style modelling provides a great introduction to data modelling because the physical and logical models are often very similar to the conceptual model. This allows the user to model complex problems in a very 'human' way, eschewing complex procedures such as de-/normalization. To demonstrate this, the following subsections introduce conceptual modelling, and illustrate how a simple model of the 'Twitterverse' can easily be mapped to a graph database management system using the property graph model.

The Conceptual Model

The conceptual data model describes a data set by identifying and mapping the main

concepts of the domain associated with the data. These concepts are often referred to as *entities*, which represent general categories, or types of data present within the domain of the data set. In turn, entities are typically associated with one another through one or more types of *relationships*, which represent interactions or associations between entities. Furthermore, entities and relationships can have attributes, which describe the characteristics of entity and relationship classes. The concept model is usually expressed as a simple chart resembling a flow chart, or concept map that highlights possible entities, their attributes, and how they are related. The conceptual model is typically the first step in data modelling and database implementation, and is crucial for high-level communication of both architectural specifications and analytical procedures.

In order to create a conceptual model, we first identify the entities present within the data set. An entity can be defined as a reference to 'a thing in the real world with an independent existence' (Elmasri and Navathe, 2010: 203). This thing could be something tangible, like a building, person, or automobile, or something intangible, like a song, a scientific theory, or political party. A good general rule when looking for entities is that they are typically things named with nouns. Using the example of Twitter, both *users* and *tweets* could be identified as examples of entities within the data set. Each of these entities would then be assigned attributes: users have attributes such as username and location, tweets have time and text, etc. After identifying the relevant entities within the data set, the task becomes determining how they are associated to one another by relationships.

Relationships are generally not things, but instead represent the type of associations found between entities. We can define a relationship type as a 'set of associations – or a relationship set – among entities' (Elmasri and Navathe, 2010: 209). These associations define the semantics of the data model in that they illustrate the logical connection between entity types. As opposed to entities, relationships are usually identified using verbs. Again looking at Twitter, an example of a possible relationship would be *tweets*, as in *user tweets tweet.* In this example, *tweets* specifies a possible relationship between *user* and *tweet* entities, but it is important to recognize that there can be more than one type of relationship between two entity types. For example, a *user* could also *favorite* a *tweet*, thus creating a new possible relationship: *user favorites tweet.*

Example: Modelling Activities in Twitter

To further illustrate the idea and practice of conceptual modelling, this example creates a conceptual model using the Paris Attacks Twitter dataset mentioned above. The process of conceptual modelling often begins with a simple description of the domain to provide a contextual framework through which data can be interpreted. The following is a high-level description of the Twitter domain that provides a starting point for the conceptual modelling process.

- In the Twitter, users can follow, or be followed by, other users.
- Tweet activity is driven by users. Users are responsible for creating Tweets, the fundamental unit of information exchange. Users have a variety of associated personal data, including their username and other optional information, such as language or geolocation.
- Tweets contain the text produced by users. They have a variety of metadata associated with their production, including location and time stamp. Furthermore, they contain semantic information that is relative to other tweets and users, such as hashtags, user references, and retweets.

While the above description is admittedly a simplified version of the activites associated with Twitter, ignoring details like attached images and videos, favorites, and personal messages, it is sufficient for our purpose in that it provides the basis for understanding

the domain. From the above description we can determine that the activity in Twitter is driven by two primary entities: the *tweet* and the *user*. Furthermore, each of these entities will have certain characteristics of interest, including tweet's text, date, and location, as well as the screen name associated with the user that created it.

To begin the modelling process, we map these two entities and their attributes to a simple model drawn with the Dia drawing program (Dia, 1998) (Figure 9.1).

Next, we must determine how they relate to one another. In many ways, this can often be the most challenging part of data modelling, as relationships are not always obvious. Fortunately, Twitter generally has very concrete and well defined relationships based on user interactions. A quick review of the above summary reveals the following possible relationships:

- *Users follow users.* Each user has a list of associated users who they follow.
- *Users tweet tweets.* Each tweet can be directly associated with the user who created it.
- *Tweet references user.* Optionally, a tweet can contain a direct reference to another user.
- *Tweet responds to tweet.* Optionally, a tweet can be a response to another tweet.
- *Tweet retweets tweet.* Optionally, a tweet can encapsulate a previous tweet, making it a retweet.

Adding these relationships to our data model, we begin to see a more complete image of the Twitter domain that encompasses a wide variety of the semantic possibilities represented (Figure 9.2).

However, before continuing it is important to compare the domain model to the actual dataset at hand to determine if there are any missing elements, or perhaps spurious information not included in the data set.

Looking through the fields of a JSON formatted tweet provided by the Twitter Application Programming Interface (API) – a service that allows users to collect up to one percent of total tweet traffic in real time based on filtering parameter – more on APIs in Janetzkos' chapter (this volume) – we see that a wide variety of metadata about the tweet is provided, including:

- Unique ids for both tweets and users
- Geolocation information for tweets when available
- Timestamps for each tweet
- A list of entities associated with each tweet

Looking more closely at the entity lists, we see that Twitter's data model considers hashtags to be entities, not just part of the tweet's text. This raises the question of whether or not our model should represent hashtags as a unique entity type. While the answer to this question depends entirely on

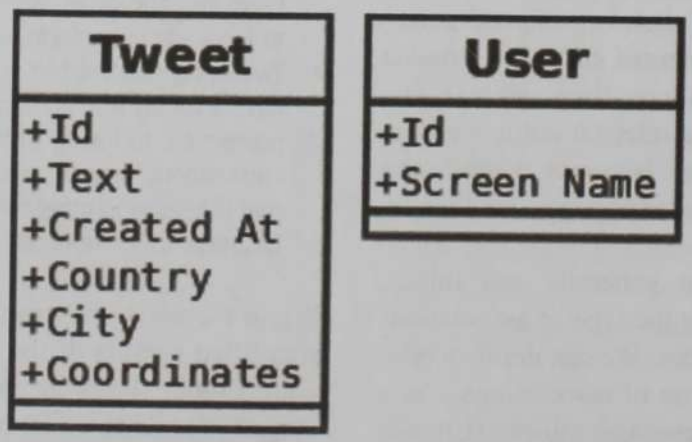

Figure 9.1 Basic Twitter entities and their attributes

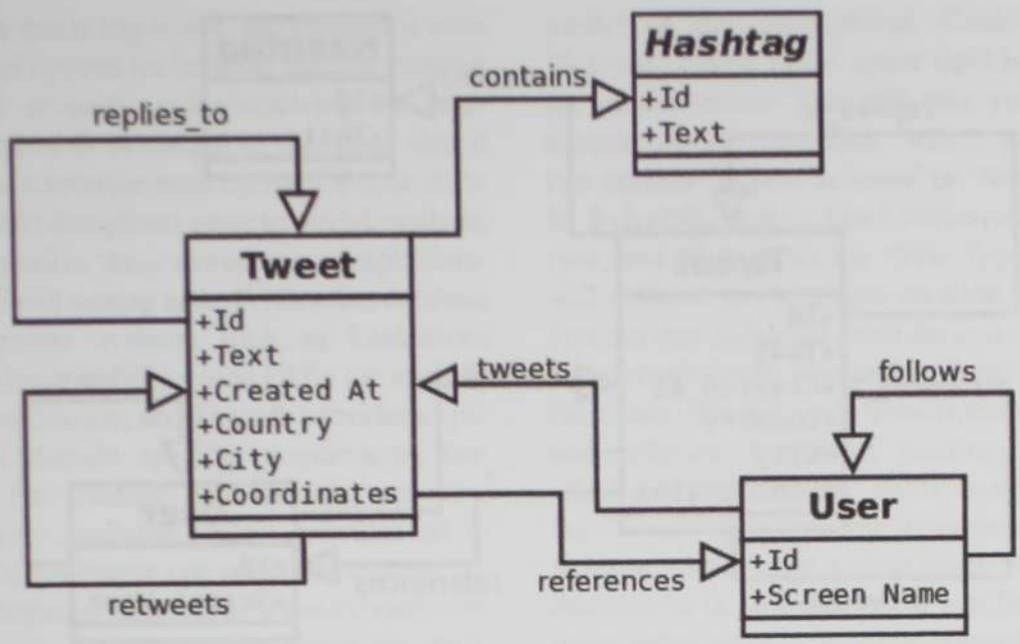

Figure 9.2 A simple conceptual model of the Twitterverse

analytic goals, it is easy to see how modelling hashtags as entities could be useful. Hashtags serve as an easy way to group tweets by thematic content without having to perform complex natural language processing (NLP) techniques such as topic modelling. Furthermore, they are user defined topic keywords, and therefore benefit from human judgment that is able to detect thematic nuances often overlooked by computers. In the case that they are not needed during analysis, hashtag entities and their relationships can simply be ignored.

Furthermore, we notice that there is no data that relates to one user following, or being followed by, another. As it turns out, Twitter's streaming API (Twitter Streaming APIs, 2010) only produces metadata relating specifically to tweets. In order to receive information about users, including follower information, one must issue specific queries to Twitter's REST API.

Therefore, the above conceptual model must be modified to reflect the semantics of the data at hand (Figure 9.3). While the original Twitter domain model that includes *user follows user* relationships is an accurate representation of activities on Twitter, it does not reflect the characteristics of the dataset. Furthermore, while hashtags were not considered in the original domain model, their presence in the data set prompted their inclusion in the data model. These sort of modifications are quite common, and this demonstrates the importance of performing multiple iterations during the modelling process.

The Property Graph Model

After the domain and dataset have been analyzed and modelled, the resulting conceptual model must be adapted to fit a database system. This is typically accomplished through an iterative process that incorporates both the logical and physical models. In this case, we will use the property graph model to create a design suitable for a graph database such as Neo4j (Neo4j, 2007) or Titan:db (Titan Distributed Graph Database, 2015). While a property graph model combines elements from both the logical and physical model, at its core it is very similar to a conceptual model in that it uses only three primitives that can be easily mapped to the three

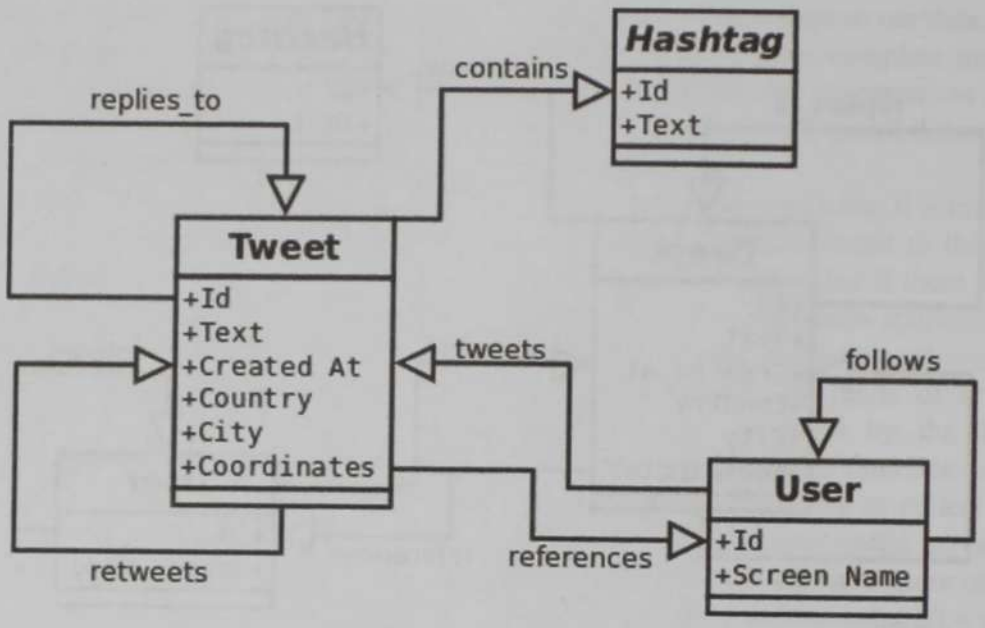

Figure 9.3 The final conceptual model of the Twitterverse

primitives used by a conceptual model (Robinson, Webber, and Eifrem, 2015):

- Properties are elements used to describe other elements. They consist of a key-value pair in which the key is a string, and the value can contain complex data types. Properties correspond to the conceptual model's attributes.
- Nodes (vertices) are elements that contain properties. Nodes typically have a type, often referred to as a label, which refers to a set of nodes that share predetermined properties. Nodes correspond to the conceptual model's entities.
- Relationships represent associations between nodes; they define the structure of the graph. Relationships often have a type, or label, that determines the semantics of a possible relationship between two node types. In a property graph, all relationships are directed in that they have a start node and an end node, which further specifies the semantics of the relationship between two nodes. Relationships can also contain properties. They correspond to the conceptual model's relationships.

In many cases, the conceptual model and property graph are identical in everything except the language used to describe them; however, the requirements of the graph database system in which the data will be stored must be taken into account. While schemaless databases like Neo4j do not require any sort of model definition prior to use, some graph database management systems, such as Titan:db, encourage the definition of a property graph style data model (schema) that includes the specification of the datatype of each property. Indeed, most database systems that implement the property graph model have some level of optional model definition designed to improve the performance or usability of the database. To illustrate this, the following section demonstrates flexible property graph style schema definition and data modelling with the SylvaDB (cite) graph database management system.

Example: Mapping Our Case-study of Twitter to a Property Graph with SylvaDB

To illustrate the process of mapping a conceptual model to a property graph, this section provides an example that utilizes SylvaDB's flexible schema creation tool to create a property graph model. SylvaDB is open source

software that is free to use, and provides a wide variety of options for building queries, running network statistics, and visualizing the database. SylvaDB is unique in the sense that it provides a browser-based graphical user interface (GUI) that allows users to model, analyze, and visualize data stored in a graph database without writing code. While other database management systems such as Linkurious (Linkurious, 2013) provide GUIs for analysis and visualization, and Titan:db provides a specialized domain specific programming language for creating data models, no other commonly available system provides all of these capabilities in one software package.

To begin, simply go to www.sylvadb.com and create a free account. From the dashboard, click on the 'New Graph' button and give your graph a name. We will use the name 'Twitterverse'. You will be redirected back to the dashboard, but you should now see your graph listed on the left hand side of the screen in the 'Graphs' column. Click on your newly created graph, and you will see a notification telling you that your schema is empty. To remedy this, we will create a new schema based on our property graph model of the Twitterverse. Click on the 'Schema' button in the upper right hand corner of the screen. This will take you to the schema builder interface, where you have two options 'Import Schema' or 'New Type'. In SylvaDB, node's label corresponds to a *type*, and clicking on the 'New Type' button will redirect you to a type creation form. As you can see, SylvaDB provides you with several default fields. Under the 'Type' heading there are 'Name' and 'Description' fields, and under the 'Properties' heading, there are 'Key' and 'Description' fields. Starting with the 'Tweet' entity, we can enter the name of the *type* 'Tweet', and a short description. The description is optional, but it can help other users better understand your schema. In this case we will simply enter 'A user's tweet as provided by the Twitter API'.

After type creation, we must create the properties associated with this type. One of the most important attributes associated with a tweet is the date and time when it was published. Therefore, we will add the 'date' as the first property. To do this we simply type 'date' under the heading 'Key' in the properties list (Figure 9.4).

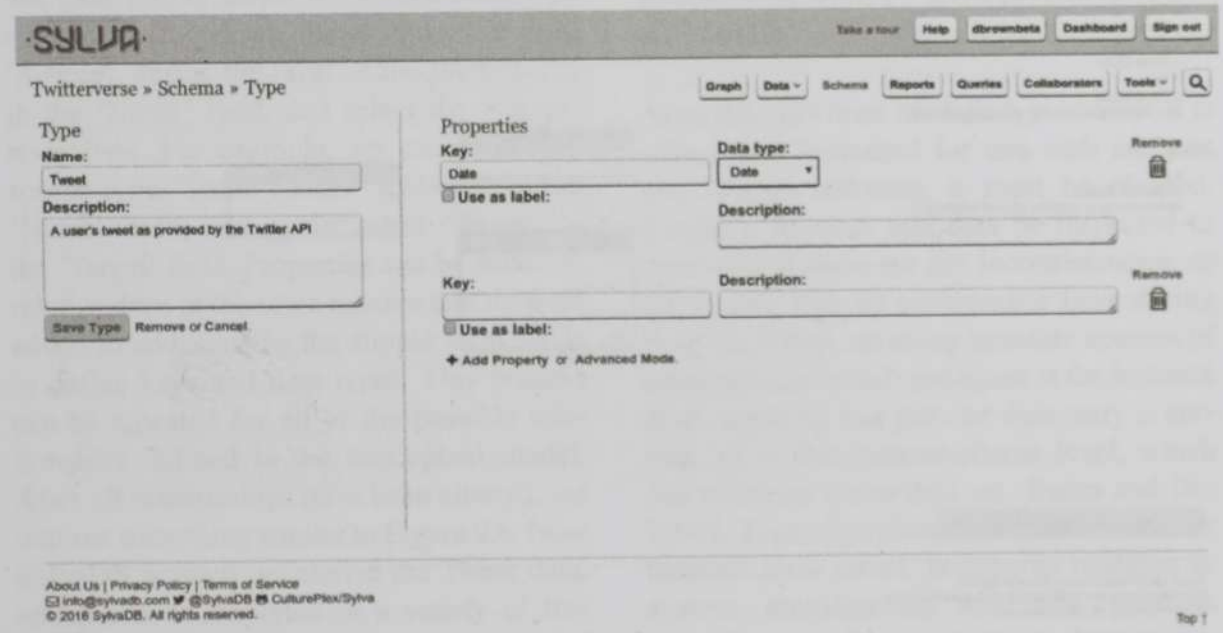

Figure 9.4 SylvaDB's type creation form

We have the option of entering a description for this property, but as date seems self-explanatory, we will leave this field blank. If we were to continue without doing anything else, the database would expect to receive a string, or a group of characters, in the date field. However, in order to leverage more advanced functionality for writing queries and performing analysis – date comparisons, advanced filtering, time series visualizations – we can instead choose to store the date in a specialized date format. To do so, we simply click on the link that says 'Advanced Mode', which expands the form field to provide a dropdown select menu that includes a wide variety of data types. Then, we simply select 'date', and SylvaDB will expect that all input for this field will be properly formatted as a date, and store it accordingly. We can continue this process, adding the other properties included in the original data model. When we have finished adding all of the attributes, we can simply click on the 'Save Type' button and the entity type will be added to the schema. In case we made an error or forgot a property, we can always go back to the *type* form and edit the information, even after there is already data in the database.

After all of the entities described in the data model have been entered as types in the SylvaDB schema builder, you should see something similar to Figure 9.5.

This means the schema is almost complete; however, it is still missing one crucial step. To finish the model, the user must define the allowed *relationships* between the different entities. Similar to defining a *type*, a *relationship* is defined by filling out a form. To access this form, a user can click the 'New Allowed Relationship' button on the schema page, or the 'new incoming' or 'outgoing allowed relationship' links that appear under the *types*.

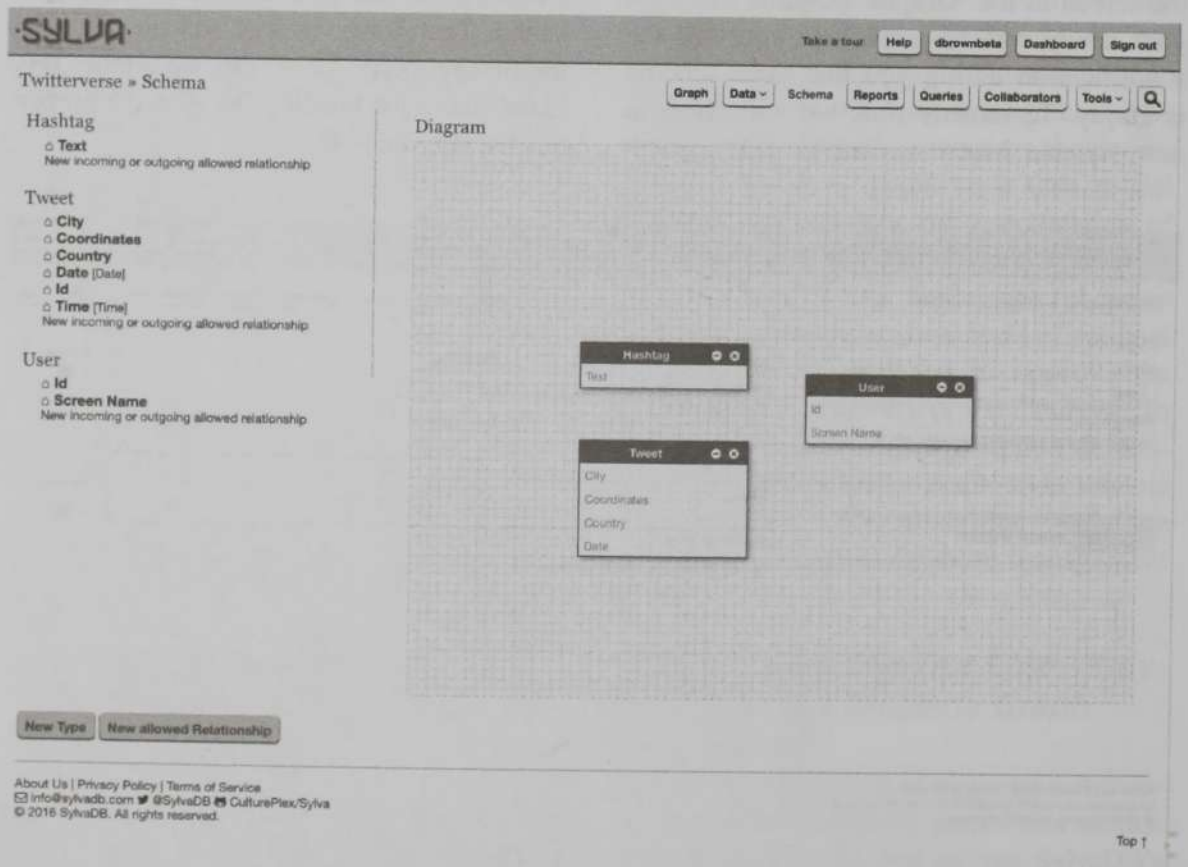

Figure 9.5 SylvaDB's property graph schema creation interface

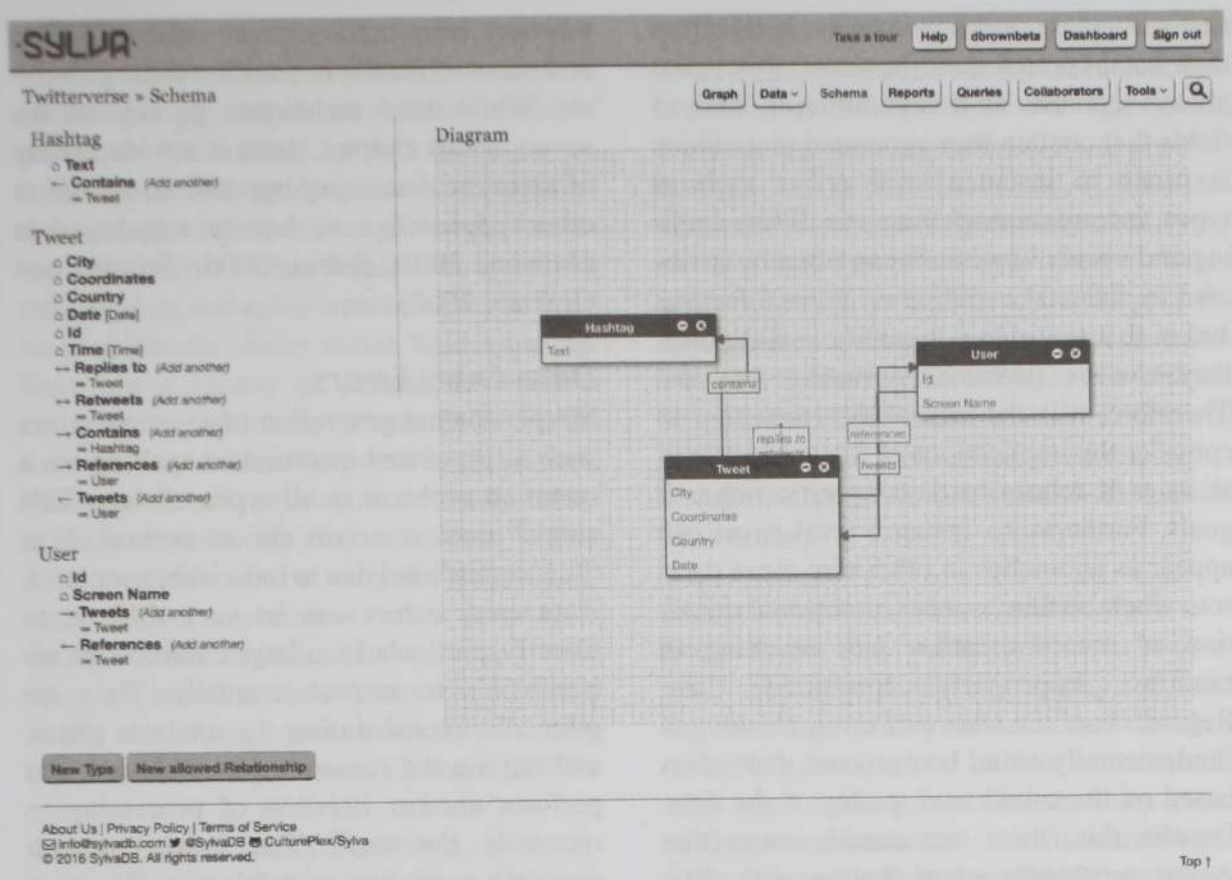

Figure 9.6 The Twitterverse conceptual model mapped to a property graph schema with SylvaDB

We then simply select the source node type using the drop down menu under the field 'Source', define the label of the relationship in the 'Name' field, and select the 'Target' node type. For example, we can select the source type 'Tweet', enter 'retweets' in the 'Name' field, and again select 'Tweet' as the 'Target' field. Properties can be added to relationships in the same manner that they are added to nodes, using the simple form fields to define keys and data types. This process can be repeated for all of the possible relationships defined in the conceptual model. After all relationships have been entered, we will see something similar to Figure 9.6. Now SylvaDB is ready to receive the Tweet data, which can be imported in a variety of formats including CSV and GEXF (a version of XML designed specifically for graph data), or entered manually.

CLEANING DATA – STANDARDIZING API DATA

After data has been modelled, but before it is parsed and formatted for use with analysis and storage software, it must be cleaned. Cleaning requires that data be inspected to determine if there are any inconsistencies, or errors that will be problematic later during analysis. There are many possible sources of errors in data, which can occur at the instance level, meaning that just one data entry is corrupt, or at the system/schema level, which can affect an entire data set (Rahm and Do, 2000). Typically, cleaning focuses more on instance level errors, as schema redesign or system modification are only possible through post collection data transformations. Due to the fact social media data is usually aggregated and stored in validated database

fields, it is less prone to instance-level errors than hand-curated data; however, this is not always the case, as it is common to analyze fields that contain user generated text, which is prone to instance level errors such as typos, inconsistent spelling, etc. While spelling and vocabulary issues can often be attributed to dialectal variation or cultural factors, and in that sense do not qualify as dirty data, they can be problematic during analysis. Therefore, it is the task of the researcher to consider the implications of this type of variation as it relates to their specific research goals. Furthermore, instance level errors can appear as anomalies in otherwise clean datasets when certain aspects of schema validation or record creation fail, resulting in random, improperly formatted data. Regardless of the source of error, cleaning is fundamentally an ad hoc process that varies based on the source and quality of the data. Despite this, there are certain issues that appear repeatedly when dealing with data. The following sections outline some of these common problems and their solutions.

Missing Data

Missing data is one of the most common problems encountered when cleaning data. Missing data can occur at the instance level, or due to lack of (or overly permissive) validation within the social media application database schema. There are a wide variety of approaches to dealing with missing data that all depend on the type of data and the researcher's goals. In many cases, data with missing fields is simply deleted or ignored, which is particularly effective with very large datasets that contain few entries with missing data. However, this technique can introduce bias and affect the representativeness of the sample, particularly if the discarded cases differ systematically from the rest of the data (Schafer, 1999). To avoid this, many statisticians use a process called imputation, which uses a variety of techniques to replace the missing data. Imputation techniques often employ advanced statistical procedures or machine learning to replace missing values and account for the imputation during analysis. While these techniques go beyond the scope of this chapter, there is a wide variety of literature discussing imputation, as well as other approaches to handle missing data (Allison, 2001; Rubin, 2004; Schafer and Graham, 2002).

Data Entry Errors

Simple content generation (data entry) errors such as typos and inconsistent spelling are a common problem in all types of data. This sort of problem occurs almost exclusively at the instance level due to individual user error. Data entry errors can be very difficult to identify, particularly in large datasets that are impossible to inspect manually. They are often discovered during the analysis phase, and require the researcher to backtrack and perform another iteration of processing to reconcile the error. The most common approach to dealing with this sort of error is to use a process called fuzzy string matching (Chaudhuri, Ganjam, Ganti, and Motwani, 2003). This technique involves calculating a distance between two strings that indicates the similarity of two entries. When multiple entries are very similar, the researcher can either manually inspect the entries to determine if they indeed refer to the same instance, or determine a maximum difference that is acceptable to programmatically resolve similar entries.

Duplicate Data

Duplicate records can occur in all kinds of datasets. While they are most common at the instance level in hand curated datasets, they can also appear in social media data – particularly data that has already undergone parsing or transformation. For example, when parsing tweet data, you may find that the ids of retweeted tweets appear hundreds or even thousands of times. While this case can be dealt with during parsing, other cases are not so straightforward. Duplicates can be difficult to diagnose, as many duplicate

entries are not recognized due to typos or missing data. Therefore, the first step in resolving duplicates often relates to the above technique of cleaning up data entry errors. After typos and spelling errors have been resolved, previously unrecognized duplicates are often visible within the dataset. However, missing data and other errors can continue to be problematic. After initial field cleaning, there are a variety of procedures used to compare the similarity of attributes across data entries to identify possible duplicates. Then, based on a minimum similarity measures determined by the researcher, highly similar entries can be merged. Many duplicate-removal techniques, also known as deduplication techniques, are also based upon advanced statistical procedures, fuzzy string matching, and machine learning (Gemmell, Rubinstein, and Chandra, 2011).

Inconsistent Units/Formats

Inconsistent use of units can also occur, especially in when combining data from a variety of sources. It is oftentimes quite difficult to identify this problem, as numeric values without specified units do not provide many clues. Resolving this sort of issue is highly dependent on the nature of the data, and common approaches are not easily delineated. For example, a wide sample of climate data from different countries may contain temperature information in both Fahrenheit and Celsius. In this case, the researcher could take into account the geographic location where the data was produced compared to the unit being used or the range of possible temperatures. This sort of error can also relate to formats, as sometimes applications store metadata such as datetimes – as specific data type that allows date and time to be precisely represented – using a format that is not compatible with other software. Schema level formatting problems are often addressed during the transformation stage; however, here we view it as part of the cleaning process because improperly formatted data are often the source of errors during data import and analysis.

These problems, whether created through data entry errors, or due to lack of constraints in the system responsible for aggregating the data, represent a small subset of possible sources of dirty data. In the end, it is the task of the researcher to determine what types of dirty data can affect their analysis based on their own goals and needs, and apply the appropriate solutions for the data at hand. Therefore, moving forward in this chapter the scope of this discussion will be narrowed to address social media data, and even further to address possible problems with generic Twitter data.

Social Media Data – Is it Dirty?

Social media data is often quite clean because it is typically produced, aggregated, and stored in high quality infrastructure based on well designed models. Indeed, social media sites like Facebook and Twitter enjoy state-of-the-art infrastructure, which translates to high standards of data quality. These standards are reflected in the data produced by their APIs, which tends to be perfectly formatted and complete; however, even complete and consistent API data can suffer from the above problems. For example, Twitter does not require geolocation data, and therefore it is common that only a small portion of API records contain coordinate data (Sloan and Morgan, 2015) . Although in this case non-geolocated tweets are allowed by Twitter's data model, and are therefore not technically dirty data, during a geographic analysis of tweets they could be considered as such. Furthermore, sometimes social media APIs produce 'dud' records: improperly formatted, partial, or otherwise impossible to parse; it is common to discover tweet records that do not contain text or user information, which can cause errors to be thrown in the parsing process. Finally, user generated text is often the most important aspect of social media data and is used for a variety of analytic tasks, many of which employ natural language processing techniques (Bifet and Frank, 2010; Kireyev, Palen and Anderson,

2009; Ronen, et al., 2014). Sometimes, in order to perform these tasks, text fields must be cleaned before they can be processed effectively. To demonstrate this, the following example employs the Twitter Paris Attacks dataset to illustrate user generated text cleaning in order to comply with later analysis requirements.

Example: Producing a Clean Tweet Field for Natural Language Processing

To illustrate field level cleaning of social media data, this example cleans the text contained within the Paris Attack tweets in order to produce a new field, *clean text,* that facilitates the application of natural language processing tasks commonly used with Twitter data such as language identification, topic modelling, and sentiment analysis. To better understand this process, as well as why it is necessary, consider the text field of the following tweet:

RT @MailOnline: 'General Curfew' ordered by French government for first time since WWII https://t.co/rk8MxzH7RT #Paris

As a human, it is relatively simple to decipher the components of this tweet:

- The RT flag identifies that this is a retweet.
- The @ symbol shows the original user that posted this tweet. Alternatively, this indicates a reference to another user as a recipient of the tweet's message.
- The # indicates a hashtag, which may or may not be part of a phrase. In this case, the word *Paris* is not included in the phrase.
- Finally, there is a link to another website, represented as a url.

After recognizing these components included with the text of the tweet, one can easily see that this tweet is written in English and that its message refers to the current situation in Paris. However, asking a computer to identify this is not as straightforward. Words beginning with characters such as # @ are not necessarily recognized as English, and URLs seem to just be strings of arbitrary characters. Furthermore, usernames are not necessarily real words, nor are they necessarily written in the same language as the rest of the tweet. Therefore, this sort of extra content in the text field can confound an algorithm designed to work with natural language. In order to make it easier for the computer to process tweet text, we can remove these sources of confusion to facilitate more accurate computer based text processing:

- The RT flag can be removed because it does not affect the content of the tweet. Instead, it indicates the endorsement of the content of a Tweet and the implied interaction between twitter users and content.
- The @ symbol, as well as the username can also be removed due to the fact that they are typically not directly related to the content of the tweet. They are, in some cases, used conversationally as proper nouns, and therefore may be included in a clean text field depending on the goals of the researcher.
- Dealing with hashtags can be a little trickier in the sense that they can be part of a phrase, or can be added to the tweet arbitrarily to indicate the content of the tweet. Due to the fact that they are generally real words and are often relevant to the content of the tweet, we will simply remove the #, while retaining the actual word used for the hashtag.
- Finally, urls will be removed as they are not natural language as such, and in that sense are not relevant to the language or sentiment associated with the tweet.

The produced *clean text* field will be considerably more readable for a computer. While this process can never be perfect when performed at a massive level, it will result in much better results than processing the raw tweet.

There are a variety of open source and commodity software packages that can be used to perform the above changes, including the commonly used Microsoft Office Excel and Libre Office Calculate, as well more specialized tools such as OpenRefine (OpenRefine, 2011). In this example, we will use the OpenRefine, which provides a wide

range of functionality for cleaning and transforming large datasets. After downloading and starting OpenRefine as specified by the documentation for your operating system, we can create a new project simply by selecting the file containing the tweet data. OpenRefine will then request that we configure parsing options, and we can simply select the field that we are interested in working with: 'text'. OpenRefine then converts the field into what looks like a spreadsheet column, and we can continue by clicking 'Create Project' in the upper right hand portion of the browser. We can then create a custom text facet – the construct used by OpenRefine to perform operations on columns of text – for the column by clicking 'Text' > 'Facet' > 'Custom Text Facet' (Figure 9.7).

OpenRefine allows the user to write custom facets using a proprietary scripting language, General Refine Expression Language (GREL), Jython, or Clojure. In this example, we will use GREL, but any of these languages would work equally well. To clean this field, we will use a series of replace methods, which allow the user to search for substrings using regular expressions and replace them as needed. For more information about regular expressions, please refer to the OpenRefine documentation. To remove the unwanted elements from the tweets, we can use a series of four chained replace methods:

1 The first method, replace("#", ""), searches the the string for the "#" character. In the case that it finds this character, it is replaced with '""', an empty string, thereby removing it.
2 The second method, replace(/RT\s/, ""), looks for the characters "RT", followed by a mandatory space (designated by the symbol "\s"). In the case that it finds this sequence of characters, they are replaced by '""', an empty string.
3 The third method, replace(/http\S*/, ""), looks for the sequence "http", followed by any non-space character, designated by "\S". Furthermore, the non-space characters can be repeated, designated by the "*". This specifies that any string beginning with "http" followed by any characters up until a space should be replaced with '""', an empty string.
4 Finally, using a similar regular expression to the one used in step 3, the fourth replace looks for any string starting with "@", again followed by any sequence of non-space

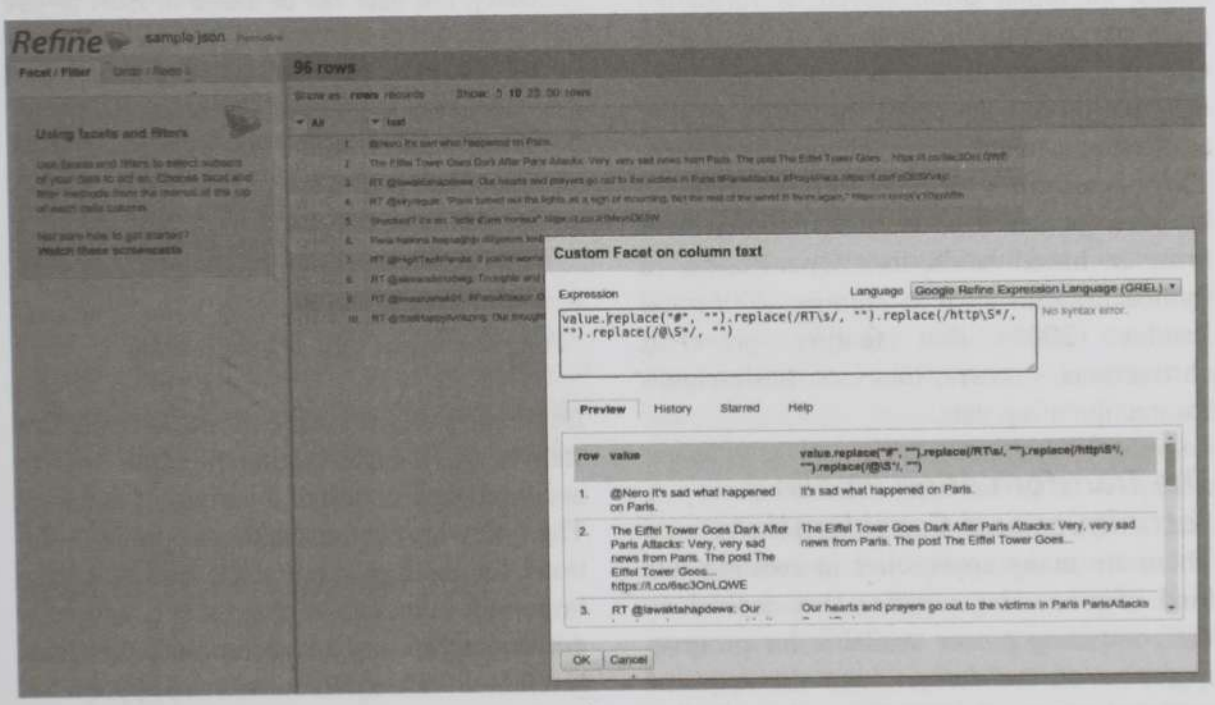

Figure 9.7 OpenRefine's custom facet creation tool

characters. If found, this string is replaced with an empty string.

Chaining together the four replace methods with the appropriate regular expressions results in the following expression:

```
value.replace("#", "").replace(/RT\s/, "").replace(/
http\S*/, "").replace(/@\S*/, "")
```

The unwanted characters have been removed from the tweet text and can be exported in one of many formats and used however necessary.

DATA TRANSFORMATION

After the dataset is modelled and cleaned, it is ready for the final stage of data processing: transformation. While the term *data transformation* can encompass a wide range of procedures, in the context of this chapter it refers to receiving input data and formatting it in a way that complies with a data model and can be imported into storage and analysis software. Essentially, the process consists of taking an input, often stored as formatted plain text, parsing the input by extracting the relevant information, and outputting the information in a specified format by writing to files or a database management system. There is a wide variety of software that can be used for this kind of task, ranging from browser based tools like OpenRefine to Python libraries such as Numpy (2006) and Pandas (2008) that feature powerful abstractions – arrays, matrices, hashmaps – for manipulating data.

The Transformation Pipeline: Techniques and Considerations

There are many approaches to creating this kind of processing pipeline that depend on the computing power available for processing, size of the dataset, and the required output; however, the transformation pipeline can be broken down into the three phases mentioned above: reading input data, parsing and transformation, and writing to an output. While each of these steps require ad hoc procedures, again there are general concerns that are relevant for all data sets. Here we broadly outline these concerns as well as potential approaches to a variety of situations.

Reading the Data: Sources and Approaches

To begin the processing pipeline, one must first consider the source and initial format of the dataset. Broadly, there are three possibilities:

- The data has not yet been acquired and it will be read directly from an API. This means that the data is still being stored by the social media application in which it was created. Typically, this kind of data is accessed programmatically and either written to an intermediate format like text files, or read dynamically from the API and processed 'on the fly'. This means that it will not be stored in any intermediate format and will be parsed, transformed, and written to output as it is produced by the API.
- The data has already been harvested from the API and is stored in flat text files. Most commonly, this data will be stored in JSON or CSV format, which can be read, parsed, and output using a wide variety of software, including all major programming languages.
- The data has already been harvested from the API, but was loaded into a database management system. Similar to data coming directly from a social media application API, relevant data will need to be read from the database and either processed on the fly, or stored in an intermediate format such as a flat text file.

Depending on the source of the data, the first step in the transformation pipeline will be accomplished in one of a variety of fashions. The following sections address possible solutions for reading input data; however, it is important to recognize that there is generally not one right way to accomplish this task. Instead, there are numerous valid approaches to this procedure that depend on the preference of the researcher.

Reading Data from an API

Typically, data is harvested from an API using one of many programmatic techniques. Each API will feature a specific architecture and protocol that provide a means to access the data it contains. Most commonly, social media site APIs use REST, an architectural style that typically uses the HTTP protocol (Massé, 2011). This is convenient because it provides a standard way to access data; although each social media site will have a unique way of providing the application data, the means for accessing is similar to other sites that use REST. This makes it easy for users to access data without having to learn specialized protocols, or use overly specialized tools.

REST resources are typically accessed using programmatic techniques. Popular programming languages like Python or Java provide a wide variety of built-in or third party software designed to facilitate the use of HTTP for accessing services like REST APIs. Generally, a researcher will write a small program specifically tailored to the target API that has been designed to harvest the data of interest. This program will then either pass the data to another program to finish the processing pipeline, or write the data to an intermediate format like flat text files. Even though writing data to an intermediate format adds an extra step to the processing pipeline, it is common to separate data collection from data processing. Furthermore, performing collection separately minimizes the moving parts involved with the pipeline; if there is a problem during the parsing or output phase, it will not affect data collection, thereby simplifying the process by compartmentalizing it.

Reading Data from Text Files

Reading data from text files is often considerably simpler than reading from an API. In many cases, it is as easy as choosing an application that is able to read the text file format. For example, there are a wide variety of desktop/browser applications that read files stored in CSV format, like Microsoft Excel, SPSS statistical software, and OpenRefine. However, depending on the operation that will be performed upon the data, these programs can be limited due to their lack of flexibility – they only provide specific hard coded procedures. Some software, such as OpenRefine, provide a balance in that they have a point and click user interface, but they also support limited scripting operations that allow the user to implement custom functionality.

In the case that the researcher needs complete flexibility (or they are comfortable with computer programming), flat text files can also be read programmatically using a wide variety of freely available programming languages. In this case, files are read and referenced by variables. Then the researcher can implement any procedure, or manipulation of the data without being limited by out-of-the-box functionality. The examples presented later in this chapter demonstrate this using the Python programming language to read and manipulate flat files.

Reading Data from a Database

In many ways, reading data from a database is similar to reading data from a web API: usually programmatic techniques are employed and relevant data is often transferred to flat files as an intermediate step before parsing and transforming the data. However, unlike a REST API, databases do not necessarily have similar architectural styles or protocols for data access. For example, some databases provide REST endpoints similar to an internet application API, others use websockets for duplex style communication between the database server and client program, and many use specialized TCP based protocols. Despite this inconsistency, most major programming languages have third party libraries that can be used to access

popular databases without delving into the specifics of the database communication protocol. In this case, the researcher must choose an appropriate library from their favorite programming language, familiarize themselves with the library, and write a normally small program to access the desired data.

Beside this type of client-server communication, many databases provide a command line interface (CLI) that allows the user to manipulate and retrieve data. Typically, a database will employ a domain specific language (DSL), or small programming language that has been custom designed to perform only the operations possible with a particular database. Fortunately, many databases share a DSL that allows a user familiar with the DSL to use a variety of database software. For example, structured query language (SQL) is a DSL used by most relational databases such as MySQL, PostgreSQL, and Oracle RDBMS (Date and Darwen, 1997).

Finally, some database management systems provide a graphical interface that allows users to access data without doing any programming. While this scenario is less common, it is important in that it provides a way for researchers without the time or means to learn computer programing access to powerful data manipulation techniques. SylvaDB, seen in a previous example, is a good representative of this kind of system. Other examples of GUI based database management systems include commonly known tools such as Microsoft Access and MySQL workbench.

Practical Considerations: Size of Data Set vs. Computational Resources

Regardless of where the input data is stored, before determining how data will be parsed it is important to consider the size of the dataset and how this will affect the parsing process. Fundamentally, there are two scenarios: 1) the dataset is small and can fit in a computer's memory (RAM); 2) the dataset is too large and cannot be loaded into memory in its entirety. Of course, whether a dataset is considered to be large depends directly on the computational resources available for processing. Therefore, it is important to understand both the size of the dataset and the amount of RAM available for processing, as well as how much extra RAM will be required for running any necessary software, or performing transformations in memory. The latter is especially important, as it is tempting to think that 4 gigabytes of data can be processed with 4 gigabytes of RAM. In reality, this is not the case, because many operations require copying data, at least temporarily, hence requiring more memory. That said, in the case that the data is too big to load into memory there are still many options that allow the researcher to handle big datasets with relatively limited computational resources.

Approaching this problem programmatically, it is common for data to be parsed on the fly. Most programming languages allow files to be opened without loading their entire contents into memory. Then, the file can be read line by line, only loading a minimal chunk of data into memory, which is parsed and written to some sort of output. Furthermore, there are data processing tools, such as OpenRefine, that use internal programmatic constructs to be memory efficient. This allows the user to perform complex operations on large data sets with a relatively small amount of RAM without writing code. However, while on the fly programmatic parsing can be performed on enormous datasets and is only limited by hard disk space and time, most browser based or GUI style software has either fixed or practical limits. For example, OpenRefine has no fixed limits, but begins to suffer performance losses on CSV files containing more than 100,000 rows of data, both due to the time complexity of the algorithms it employs and RAM limitation of typical computers. Therefore, larger datasets are typically dealt with programmatically using a scripting language such as Python.

Parsing

Regardless of the size of the dataset, social media data generally requires some degree of parsing and transformation before it can be stored and analyzed. Parsing involves dividing the data into manageable or semantically cohesive chunks, extracting relevant information, and outputting it in a specific format. In general, parsing techniques are tightly coupled to the format and semantics of the data. Due to the extremely ad hoc nature of parsing, it is more effective to present an example of the parsing process instead of simply describing it. The following section provides a concrete example of how data is parsed using the Paris Attacks Dataset and the Python programming language.

Example: Parsing a List of Tweets with Python

Parsing a list of tweets is a process that can be accomplished using any major programming language (R, Java, Perl, etc.). This example employs one of the most versatile and widespread open source programming languages: Python. To begin, we identify that the input data is stored as a list of JSON serialized tweets in flat text files. Furthermore, we assume that the computer used has plenty of RAM to store the contents of the dataset in memory; however, in this example we do not load the whole dataset. Instead, we parse the tweets on the fly, storing relevant information in data structures that will later be written to files using the TSV format. TSV, like CSV, is similar to an Excel spreadsheet in that it stores data in tabular format with rows and columns. However, instead of using commas like a CSV, it separates entries within a row using the tab character, which is more space efficient and tends to import more smoothly into certain data management systems. The target output of this process is four TSV files:

1. The first file will be a list of users and related metadata. This list can be thought of as a list of user nodes that will be mapped to a property graph model and stored in a graph database.
2. The second file will be a list of tweets and related metadata. This list represents tweet nodes that comprise the second node type of the property graph model, which will also be stored in a graph database.
3. The third file will be a list of hashtags. This is the third node type in the property graph model, representing the final node type in the graph database.
4. The fourth file will be an edge list containing edges of four types: *user tweets tweet, tweet retweets tweet, tweet replies to tweet,* and *tweet contains hashtag.* This list represents the relationships included in the property graph model, and will be used to structure the information stored in the graph database.

Using Python, we can create these four files using the built in csv module. Furthermore, we will load the json module, which will be used later to parse the JSON formatted tweets.

```
import csv
import json
tweetfile = open("tweets.tsv", "wb")
tweet_writer = csv.writer(tweetfile, delimiter="\t")
userfile = open("users.tsv", "wb")
user_writer = csv.writer(userfile, delimiter="\t")
hashtagfile = open("hashtag.tsv", "wb")
hashtag_writer = csv.writer(hashtagfile,
delimiter="\t")
edgefile = open("edges.tsv", "wb")
edge_writer = csv.writer(edgefile, delimiter="\t")
```

Using the csv module writer object, we can write data to csv files. To begin this process, we can create headers for each file. These headers specify the contents of each column in the TSV files that have been created.

```
tweet_header = ["tid", "lang", "text", "created_
at", "country", "city", "coordinates"]
tweet_writer.writerow(tweet_header)
user_header = ["uid", "screen_name"]
user_writer.writerow(user_header)
hashtag_header = ["hid", "text"]
hashtag_writer.writerow(hashtag_header)
edge_header = ["source_id", "target_id", "type"]
edge_writer.writerow(edge_header)
```

These headers can serve as a guide during the parsing process, as they determine what data needs to be extracted from the tweet. Notice that the edge header also includes the column header 'type', which will allow us to distinguish between different relationship types. After inspecting the contents of a tweet record, we also notice that there are possibly two tweets contained within each record: if the tweet is a retweet, it also includes the metadata of the original tweet. In order to avoid code duplication, we will write a simple function that extracts data from the original tweet record that can also be used on the embedded retweet record.

```
def parse_tweet(tweet):
tweet = json.loads(tweet)
tid = tweet["id"]
lang = tweet["lang"]
text = tweet["text"]
created_at = tweet["created_at"]
place = tweet.get("place", {})
country = place.get("country", "")
city = place.get("full_name", "")
coordinates = place.get("bounding_box", {}).
get("coordinates", "")
user_mentions = tweet.get("entities", {}).
get("user_mentions", [])
hashtags = tweet.get("entities", {}).
get("hashtags", [])
uid = tweet["user"]["id"]
screen_name = tweet["user"]["screen_name"]
replies_to = tweet["in_reply_to_status_id"]
retweeted_status = tweet.get("retweeted_
status", "")
return (
tid, lang, text, created_at, place, country, city,
coordinates, user_mentions,
hashtags, uid, screen_name, replies_to,
retweeted_status)
```

We will then open the tweet file, iterate over all of the tweets in the file, call this function on each tweet, and store the results as a row in a Python dictionary containing all of the tweet data. If the tweet was a retweet, we will store the retweet data in the same dictionary. User and hashtag data are stored in separate dictionaries, as they will be written to a different output file. Finally, any edges (user tweets, user mentions, in reply to, retweets, contains) will be written directly to the edge file.

```
tweet_dict = {}
user_dict = {}
hashtag_dict = {}
hashtag_id = 0
with open("paris_tweets.json", "rb") as f:
for tweet in f:
results = parse_tweet(tweet)
# basic tweet data
tid = results[0]
if tid not in tweet_dict:
row = [results[1], results[2], results[3], results[4],
results[5], results[6], results[7]]
tweet_dict[tid] = row
user_id = results[10]
if user_id not in user_dict:
user_dict[user_id] = rt_results[11]
edge_writer.writerow([user_id, tid, "TWEETS"])
# user mention data
user_mentions = results[8]
for user_mention in user_mentions:
uid = user_mention["id"]
screen_name = user_mention["screen_name"]
if uid not in user_dict:
user_dict[uid] = screen_name
edge_writer.writerow([tid, uid, "MENTIONS"])
# hashtag data
for hashtag in results[9]:
hashtag = hashtag["text"].lower()
if hashtag not in hashtag_dict:
hid = "h{}".format(hashtag_id)
hashtag_dict[hashtag] = hid
edgewriter.writerow([tid, hid, "CONTAINS"])
hashtag_id += 1
# replies to data
replies_to = results[12]
if replies_to:
edge_writer.writerow([tid, replies_to, "REPLIES_TO"])
if replies_to not in tweet_dict:
tweet_dict[replies_to] = ["", "", "", "", "", "", ""]
```

```
# retweet data
if results[-1]:
rt_results = parse_tweet(results[-1])
rt_tid = rt_results[0]
if rt_tid not in tweet_dict:
rt_row = [rt_results[1], rt_results[2], rt_results[3],
rt_results[4],
rt_results[5], rt_results[6], rt_results[7]]
tweet_dict[rt_tid] = rt_row
user_id = rt_results[10]
if user_id not in user_dict:
user_dict[user_id] = rt_results[11]
edge_writer.writerow([results[0], rt_tid,
"RETWEETS"])
edge_writer.writerow([user_id, rt_tid, "TWEETS"])
```

Finally, we write the contents of the tweet, user, and hashtag dictionaries to TSV files, and then close the original input files.

```
for k, v in tweet_dict.items():
row = [k] + v
tweet_writer.writerow(row)
for k, v in user_dict.items():
user_writer.writerow([k, v])
for k, v in hashtag_dict.items():
hashtag_writer.writerow([v, k])
tweetfile.close()
userfile.close()
hashtagfile.close()
edgefile.close()
```

Now the list of tweets has been parsed into three separate TSV files that will be easy to load into most graph database systems and analysis software. To quickly demonstrate this, the next example loads the produced files into the Neo4j graph database using the Neo4j bulk loader CLI (Neo4j-import, 2015).

Bulk Loading TSV Files into Neo4j

Provided that data has already been formatted as a series of node lists and edge lists, we can use the Neo4j bulk import tool. However, there are certain changes that must be made to the TSV files produced in the previous example in order to prepare the data for import. Thankfully, only the headers need to be changed; can be done using the Python Pandas package. We will read the files one by one and reassign certain column header names so they comply with Neo4j's specifications. Specifically, all nodes require a unique id column denoted by the: ID postfix as well as a column for label, denoted as: LABEL. Edges require a column with: START_ID, which is the source of the relationship,: END_ID, which is the target of the relationship, as well as: TYPE.

```
import pandas as pd
tweets = pd.read_csv("tweets.tsv", sep="\t")
tweets.columns = ["tid:ID", "lang", "text",
"created_at", "country", "city", "coordinates"]
tweets[":LABEL"] = "tweet"
tweets.to_csv("neo4j_tweets.tsv", sep="\t")
users = pd.read_csv("users.tsv", sep="\t")
users.columns = ["uid:ID", "screen_name"]
users[":LABEL"] = "user"
tweets.to_csv("neo4j_users.csv", sep="\t")
hashtags = pd.read_csv("hashtags.tsv", sep="\t")
hashtags.columns = ["hid:ID", "text"]
hashtags[":LABEL"] = "hashtag"
hashtags.to_csv("neo4j_hashtags.tsv", sep="\t")
edges = pd.read_csv("edges.csv", sep="\t")
edges.columns = [":START_ID", ":END_ID", ":TYPE"]
edges.to_csv("neo4j_edges", sep="\t")
```

After preparing the data set, one must install and unpack Neo4j, navigate to the root directory (something like neo4j-community-2.3.1/), and use the command line import tool to load the data. With the command line tool, we have to specify the destination directory where the data will be stored (by default/data/graph.db), each node list that will be imported, the edge list that will be imported, and the delimiter used for the files.

```
./bin/neo4j-import –into/neo4j-community-2.3.1/
data/graph.db –nodes neo4j_users.tsv –nodes
neo4j_tweets.tsv –nodes neo4j_hashtags.tsv –
relationships neo4j_edges.tsv –multiline-
fields=true –delimiter TAB
```

This command can be broken down as follows:

- The main command, "./bin/neo4j-import", runs the import executable included in the Neo4j database distribution.
- The –into argument specifies the destination directory for the processed output. This is where the data is stored and accessed by Neo4j. With Neo4j's default configuration, this directory should be "data/graph.db".
- The –nodes arguments are used to specify the names of the files that contain the data that will be imported to create nodes.
- The –relationships arguments are used to specify the names of the files that contain the data that will be imported to created relationships.
- The –multiline-field argument determines whether or not the input fields can contain newline characters ("\n"). Since tweet text can contain newlines, if this argument is not specified as true, the import will throw errors.
- Finally, the –delimiter argument specifies the character used to separate the entries in the input files. This argument value defaults to a comma, but because we are using TSV files, we indicate that this value should be a tab "\t".

After running this command and waiting for the data to be imported, we can start the Neo4j server, and begin writing queries using Neo4j's expressive graph query language: Cypher (Cypher Query Language, 2012).

CONCLUSION

As we have seen, processing social media data requires a wide variety of techniques and a broad range of skills. Fortunately, there are a wide range of tools, both programmatic and GUI based, that are specifically designed to work with this kind of data. As social media becomes even more prevalent, the number of individuals seeking to leverage the wealth of data provided by users will surely grow. As more and more researchers – in both academia and industry – dedicate themselves to studying this data and producing actionable information, the range and quality of techniques and tooling will increase. While no individual can be expected to master all of the software dedicated to this sort of data processing, this chapter demonstrates that despite the typically one-off nature of data processing, there are certain commonalities that span the range of possible data sets. Regardless of how big or small a dataset may be, whether it be rife with errors, or sparkling clean, to achieve satisfactory results all data must be modelled, assessed for cleanliness and field formatting, and parsed into a format that is compatible with target storage and analysis software. We hope that after reading this chapter you will feel more comfortable taking charge of your data to produce the best results possible.

REFERENCES

Allison, Paul D. (2001). *Missing Data* (Vol. 136). Thousand Oaks, CA: Sage.

Bifet, Albert and Frank, Eibe. (2010). 'Sentiment knowledge discovery in twitter streaming data', *Discovery Science*: 1–15. Berlin: Springer.

Chaudhuri, Surajit, Ganjam, Kris, Ganti, Venkatesh and Motwani, Rajeev. (2003). 'Robust and efficient fuzzy match for online data cleaning', *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 313–324.

Cypher Query Language. (2012). Retrieved from http://neo4j.com/docs/stable/cypher-query-lang.html

Date, Chris J. and Darwen, Hugh. (1997). *A Guide To Sql Standard* (Vol. 3). Reading: Addison-Wesley.

Dia [computer software]. (1998). Retrieved from https://sourceforge.net/projects/dia-installer/

Elmasri, Ramez and Navathe, Shamkant. (2010). *Fundamentals of Database Systems* (6th ed.). Boston: Addison-Wesley Publishing Company.

Kireyev, Kirill, Palen, Leysia and Anderson, Kenneth. (2009). 'Applications of topics models to analysis of disaster-related twitter data', *NIPS*

Workshop on Applications for Topic Models: Text and Beyond (Vol. 1). Canada: Whistler.

Gemmell, Jim, Rubinstein, Benjamin, and Chandra, Ashok. (2011). Improving entity resolution with global constraints. *arXiv preprint arXiv:1108.6016*.

Linkurious [Computer software]. (2013). Retrieved from http://linkurio.us/

Massé, Mark. (2011). *REST API design rulebook*. Sebastopol, CA: O'Reilly Media, Inc.

McCulloh, Ian, Armstrong, Helen, and Johnson, Anthony. (2013). *Social network analysis with applications*. John Wiley & Sons.

Neo4j [Computer software]. (2007). Retrieved from http://neo4j.com/

Neo4j-import- [Computer software]. (2015). Retrieved from http://neo4j.com/docs/stable/import-tool.html

Numpy [Computer software]. (2006). Retrieved from http://www.numpy.org/

OpenRefine [Computer software]. (2011). Retrieved from http://openrefine.org/documentation.html

Pandas [Computer software]. (2008). Retrieved from http://pandas.pydata.org/

Rahm, Erhard, and Do, Hong Hai. (2000). 'Data cleaning: Problems and current approaches', *IEEE Data Eng. Bull.* 23.4: 3–13.

Rainie, Lee and Wellman, Barry. (2012). *Networked: The new social operating system*. Cambridge, MA: MIT Press.

Robinson, Ian, Webber, Jim and Eifrem, Emil. (2015). *Graph Databases: New Opportunities for Connected Data*. Sebastopol, CA: O'Reilly Media, Inc.

Ronen, Shahar, Gonçalves, Bruno, Hu, Kevin Z., Vespignani, Alessandro, Pinker, Steven, and Hidalgo, César A. (2014). 'Links that speak: The global language network and its association with global fame', *Proceedings of the National Academy of Sciences*, 111(52): E5616-E5622.

Rubin, Donald B. (2004). *Multiple imputation for nonresponse in surveys* (Vol. 81). John Wiley & Sons.

Schafer, Joseph L. (1999). 'Multiple imputation: a primer'. *Statistical Methods in Medical Research*, *8*(1): 3–15.

Schafer, Joseph L. and Graham, John W. (2002). 'Missing data: our view of the state of the art', *Psychological Methods*, *7*(2): 147.

Sloan, Luke and Morgan, Jeffrey. (2015). 'Who Tweets with Their Location? Understanding the relationship between demographic characteristics and the use of geoservices and geotagging on Twitter'. *PloS one*, *10*(11), p.e0142209.

Titan Distributed Graph Database [Computer software]. (2015). Retrieved from http://thinkaurelius.github.io/titan/

Twitter Streaming APIs. (2010). Retrieved from https://dev.twitter.com/streaming/overview