PharmaSUG 2017 - Paper AD25

Controlled Terminology without Excel

Mike Molter, Wright Avenue Partners

ABSTRACT

The world of clinical data standards brought with it plenty of rules about metadata. This domain must have these variables, those variable values must be no longer than 8 characters, these other variables are restricted to certain values, etc. What standards came up short on was how/where these rules could be stored and maintained, as well as any processes for implementing them. Nowhere is this more evident than controlled terminology.

Standard metadata such as NCI controlled terminology has been made available through Excel, and the rules for implementing these standards are stored in PDF documents. Much of the industry has defaulted to Excel for the collection and maintenance of study metadata. While on the surface, copying and pasting codelists from the NCI spreadsheet to a study spreadsheet may seem straightforward, Excel doesn't have the capability to enforce rules such as codelist extensions, sponsor-defined codelists, allowable data types, and subsetting a codelist multiple ways for multiple variables. Additionally, controlling access to a centralized Excel file has its challenges.

This paper presents a simple controlled terminology application which features storage and maintenance through a graph database. This paper will show similarities and differences between graph databases and SAS® data sets. A web interface that guides a user through defining controlled terminology for a study, following CDISC rules, without copying and pasting, and without Excel, will be demonstrated. Users will discover the value of standards stored in a database and controlled access through web forms.

INTRODUCTION

The requirement of a formal submission of metadata that describes our clinical trial data set the industry into a mild panic. After all, while some of the required metadata overlapped with that which we associate with a programming specification, some of it was metadata we had never considered before. From where would this metadata originate? Even if the define.xml was being built from already-existing data sets, not all of the metadata could be extracted from SAS metadata (i.e. output from the CONTENTS procedure). And where would we put it? We know that its final destination had to be an industry-defined XML structure, but how would we initially collect, organize, and manage it? For many, the need to convert collected data to standardized formats was enough of a burden to add to the ever-growing mountain of study work. Formal processes around metadata – data entry, storage, management - would have to be minimal, involving only current staff and software they already had and knew how to use. The quickest and most popular choice was Excel.

On the surface, Excel was an obvious choice for many reasons. For starters, everyone has it and knows how to use it. In addition, each level of metadata appeared to have a two-dimensional aspect to it. One dimension was the set of all metadata properties for that level of metadata. These could form the columns of a spreadsheet. The second dimension was whatever made that level of metadata unique, making up the rows of a spreadsheet. For example, data set-level metadata could be described with one row of a spreadsheet per data set, and one column for each metadata property used to describe it. The fact that define.xml requires multiple levels of metadata could be handled by multiple tabs in a spreadsheet (a third dimension). Of course Excel files can easily be saved to a network drive and attached to emails, making them easy to share with everyone interested. And finally, while standard

metadata for SDTM safety domains and ADaM classes were initially only published in PDF files, the CDISC SHARE team started making them available in Excel, thereby at least providing users with a head start on study metadata. NCI was doing the same with controlled terminology.

Trouble began to appear though when we started to realize that some of the flexibility mentioned above could also be its downfall. For starters, these files made available by organizations such as CDISC and NCI that contain industry requirements are available for download to anyone in an organization (in the case of CDISC, anyone with membership credentials) to any network drive. While these files become the template for building study metadata, the ease in which they can be accessed, copied, modified, even accidentally deleted makes them and ultimately the processes and final products that rely on them vulnerable. The same risks are inherent in any Excel file that contains study metadata.

The flexibility of Excel means that Excel cannot enforce the rules of define.xml. For example, Excel doesn't know that when a data set is defined, it must also have variables, or that deleting a data set definition also requires deleting its variables. Excel doesn't know that referenced codelists and dictionaries must be defined, and that defined codelists and dictionaries must be referenced at least once. Excel doesn't know that derived variables must make reference to defined methods, and that variables of type "float" must have significant digits specified. Excel wasn't written to know these rules and so we count on the user to read any user guidelines we put together and follow the rules.

A well-designed database to which access is controlled, along with software with a smart interface that knows the rules, can overcome some of these shortcomings. Such a database is one that the user never sees and has no means of manipulating, except indirectly through choices made while traversing the interface. Such choices are offered through software that knows the rules, and that provides the user with a platform for providing metadata that follows these rules. Maybe most importantly, this platform can have a profound effect on the user experience. Such a platform can provide for a more natural experience, guiding the user through a storyline of their study, rather than opening an Excel workbook, choosing an arbitrary starting point, and simply filling in rows and columns.

The subject of this paper is a small-scale application for defining controlled terminology, both at a global and a study level. The name of this application is Controlled Terminology Designer (CTD). We'll start with a review of what NCI makes available today to the public, what many are doing with it, and what is required in a metadata submission. We'll then discuss the functionality of the application and some of the technological nuts and bolts that sit behind the scenes.

CURRENT CONTROLLED TERMINOLOGY PRACTICES

Let's begin by making sure we are all on the same page with regard to what controlled terminology is. A study's controlled terminology is a set of codelists. A codelist is a set of allowable values for one or more variables. In the old days, a codelist might be informally defined as part of the database design and spec-writing processes. Today, controlled terminology is a formal part of the metadata that represents collected as well as submission data. Additionally, in today's world of standards, many codelists have required terms to which the data must adhere. These codelists are published by NCI in an Excel file that can be downloaded from their website.

The Excel file contains a mixture of header and detail rows, where the header rows, colored in blue, contain information about the codelist as a whole, and the detail rows below the header row with only the white background each provide information about a term in the codelist. Figure 1 below shows some of the columns.

Figure 1 – Sample Controlled Terminology Excel file provided by NCI

	A	B	C	D	E	F	
1	Code	Codelist Code	Codelist Extensible (Yes/No)	Codelist Name	CDISC Submission Value	CDISC Synonym(s)	CDIS
46	C50799	C101859		Cardiac Procedure Indication	VENTRICULAR FIBRILLATION		Ventricular Fibrillation is characteri (cycle length: 180 ms or less), gro marked variability in QRS cycle len
47	C101866		Yes	Cardiac Rhythm Device Failure Manifestation	CRYDFMAN	Cardiac Rhythm Device Failure Manifestation	The effect that a cardiac rhythm de stimulate the heart.
48	C99921	C101866		Cardiac Rhythm Device Failure Manifestation	ATRIAL PACING MALFUNCTION		The cardiac rhythm device malfunc
49	C99923	C101866		Cardiac Rhythm Device Failure Manifestation	DEFIBRILLATION MALFUNCTION		The cardiac rhythm device malfunc
50	C99973	C101866		Cardiac Rhythm Device Failure Manifestation	LEFT VENTRICULAR PACING MALFUNCTION		The cardiac rhythm device malfunc pacing.
51	C100011	C101866		Cardiac Rhythm Device Failure Manifestation	RIGHT VENTRICULAR PACING MALFUNCTION		The cardiac rhythm device malfunc pacing.
52	C119016		No	Cardiovascular Findings About Results	CVFARS	Cardiovascular Findings About Results	Terminology responses used for th and names codelist within CDISC.
53	C20080	C119016		Cardiovascular Findings About Results	ANGIOGRAPHY		A method used to create an image
	C119199	C119016		Cardiovascular Findings About Results	AUGMENTATION OF ORAL DIURETIC		Initiation or intensification of orally

Column E, titled “CDISC Submission Value” contains the short name of the codelist on the header rows, and the allowable values for variables on the detail rows. The long name of the codelist is in all rows of column D. NCI-assigned codes are found in columns A and B. Codes assigned to the codelist as a whole are found in column A on the header rows and column B on the detail rows. Codes assigned to each of the terms are found on detail rows of column A.

On the surface, it may seem convenient for anyone collecting and managing their define.xml metadata in Excel that NCI gives us this Excel file. To assess just how convenient it is however, we need to examine exactly what is required for define.xml.

For starters, the requirement for controlled terminology in define.xml goes beyond just the allowable values. Each codelist has a required set of metadata properties, as does each term in each codelist. Below is the set of required codelist properties.

- OID (object identifier) – OIDs are found all over the place in define.xml. Their values say nothing about the data, but rather are used to link to other metadata elements. Codelists are referenced inside variable- and value-level metadata items by way of the OID.
- Name – a description of the codelist
- Data Type – an indicator of what kind of data the codelist holds. Valid values are “text”, “integer”, and “float”.
- NCI Code – The code assigned by NCI to the codelist

Are all of these available on the blue lines of the NCI file? As mentioned earlier, the NCI code is found in column A, and the codelist description is found in column D. OID values are sponsor-defined, but out of convenience, standard convention is to use some function of the value found in column E. DataType is not provided. Most codelists in the NCI file (in some cases, all codelists) are “text” codelists, but because this isn’t stated explicitly in the file, it is the responsibility of anyone using these codelists to make this determination.

The following are the properties used to describe codelist terms.

- Coded Value – these are the actual allowable data values
- Decode – an optional description of the term
- Extended Value – Indicates that a term is not part of the NCI-defined codelist
- NCI Code – The code assigned by NCI to the term

As mentioned above, the coded value can be found in column E of the spreadsheet. Decodes are optional because sometimes the coded value carries enough information to make a decode unnecessary.

If the terms of a codelist require explanation, one can use the NCI Preferred Term from column H. The NCI code is found in column A.

In short, the columns contained in the NCI file give us most of the properties we need for documenting controlled terminology in define.xml, but our responsibilities extend beyond just the properties. For starters, the controlled terminology we document in our define.xml must contain only codelists relevant to the study being described, not all of the codelists in the NCI file. This means that we can use only a subset of the rows from the NCI file. Additionally, variable- and value-level metadata standards give us an idea of which codelist describes the variable's values in general, but oftentimes it is some modification of that codelist that is necessary at a study level. A modification of a codelist involves a combination of a subset of the NCI-defined terms for that codelist, plus values not defined by the codelist. Codelists deemed as "extensible" by NCI (as indicated in column C of the header rows of the NCI file) allow us to add to the codelist, terms that are not equivalent to any other NCI-defined terms. An NCI codelist must be subset when it contains terms that are not relevant to the study. Consider NCI's UNIT codelist. This codelist contains about 640 different units. Only a handful of these terms will apply to any given unit variable such as EXDOSEU. While the standards consist of many unit variables, all of which refer to the same UNIT NCI codelist, at the study level, the appropriate subset codelist must be defined for each of these variables.

We're starting to see that while the NCI Excel file provides a baseline for study controlled terminology, it cannot be used as the sole input for any tool that creates a study's define.xml. If we insist on using Excel to collect and manage our metadata, then part of the task of defining controlled terminology must be to pull from the NCI file the columns and rows that are relevant for our study. Another part of that task is to manually insert rows that represent terms not defined by NCI. As mentioned above, this applies to extensions of NCI codelists, but it also applies to entire codelists. The standards make clear that some variables must be associated with a codelist, but it's up to the sponsor to define that codelist. For sponsor-defined terms and codelists, it is up to the sponsor to define all of the necessary properties.

All of this points to a task that involves a substantial combination of copying and pasting as well as manual data entry. If most users know enough about Excel, then why is this a problem? For starters, time and accuracy can be an issue. One might argue that copying rows from one Excel file, switching over to another Excel file, and then pasting them doesn't take long, but compare it to simply selecting choices on a web form – choices that are informed by a database. It may not take long to determine the value of a property that NCI doesn't provide, such as Data Type, but compare that to not having to provide it at all, because it's already in a database. Inserting Excel rows to accommodate extended terms or sponsor-defined codelists might not seem to take long, but it's another task that good software doesn't require. None of these Excel tasks take long if they only have to be done once, but it adds up over the course of dozens of codelists and hundreds of terms. Through all of this busy Excel work, accuracy is bound to suffer – copying and pasting the wrong rows, entering "test" instead of "text", etc, and Excel won't be available to catch those mistakes. Maybe the most significant problem is that the accumulation of these tasks takes us off of our storyline. It's like trying to read a book, but after every read sentence, you take time to answer a text or an email or a phone call. The storyline is constantly interrupted, and the user experience is clouded with Excel noise.

The task of defining controlled terminology for a study is a long, and dare I say, tedious one. Furthermore, no software (as of now) can do it for us – no software can know how an NCI codelist needs to be extended and/or subset, or how a sponsor codelist should be defined. What it can do, and what CTD attempts to do, is to keep you on your storyline without interruptions by making sure that your line of thinking remains focused on the study, thereby minimizing administrative tasks. Users still spend time on

the controlled terminology requirements of define.xml, but the time spent implementing these decisions is minimized. After completing one controlled terminology task, the user immediately moves on to the next.

CTD – WHAT DOES IT DO AND HOW DOES IT DO IT?

CTD is an open-source, web-based application that allows users to define controlled terminology at both a global and study level. At a global level, users can define "child" codelists based on "parent" codelists that are defined by NCI. A child codelist is a modification, either through subsetting, extension (if the parent codelist is extensible), or both, of a parent codelist. Once a user selects an NCI controlled terminology version, the user is presented a list of all parent codelists from that version, plus any child codelists that have been defined (see Figure 2).

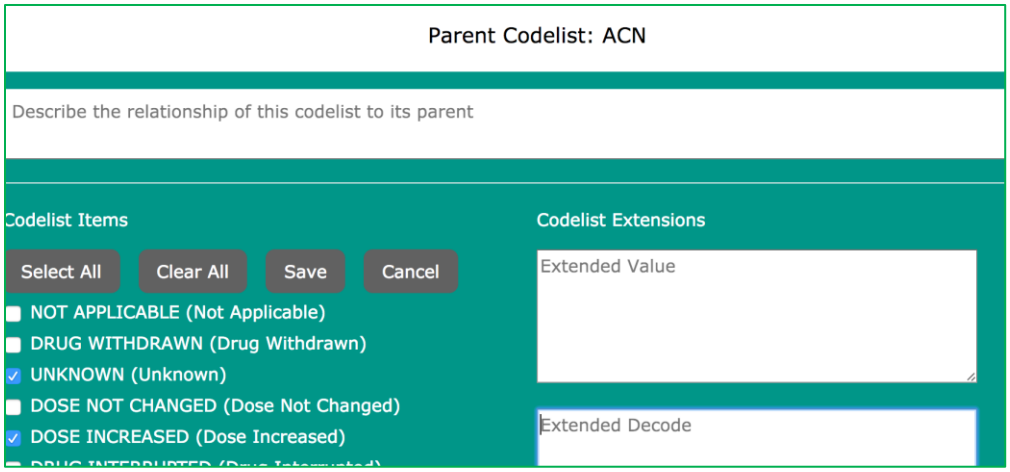
Figure 2 – List of all NCI codelists of a chosen version, plus any child codelists that have been added

NCI Controlled Terminology
Publication Version: 2016-06-24

Codelist code	Codelist	Description	Click to create a child
C103459	ACGC01TC	Alzheimer's Disease Cooperative Study-Clinical Global Impression of Change Questionnaire Test Code	Select
C103459	ACGC01TC1_NCI	Alzheimer's Disease Cooperative Study-Clinical Global Impression of Change Questionnaire Test Code - count children test	Select
C103459	ACGC01TC2_NCI	Alzheimer's Disease Cooperative Study-Clinical Global Impression of Change Questionnaire Test Code	Select
C103459	ACGC01TC3_NCI	Alzheimer's Disease Cooperative Study-Clinical Global Impression of Change Questionnaire Test Code	Select
C103458	ACGC01TN	Alzheimer's Disease Cooperative Study-Clinical Global Impression of Change Questionnaire Test Name	Select
C103458	ACGC01TN1_NCI	Alzheimer's Disease Cooperative Study-Clinical Global Impression of Change Questionnaire Test Name	Select
C66767	ACN	Action Taken with Study Treatment	Select
C66767	ACN1	Action Taken with Study Treatment - ACN TEST 1	Select
C66767	ACN2	Action Taken with Study Treatment - ACN TEST 1460	Select

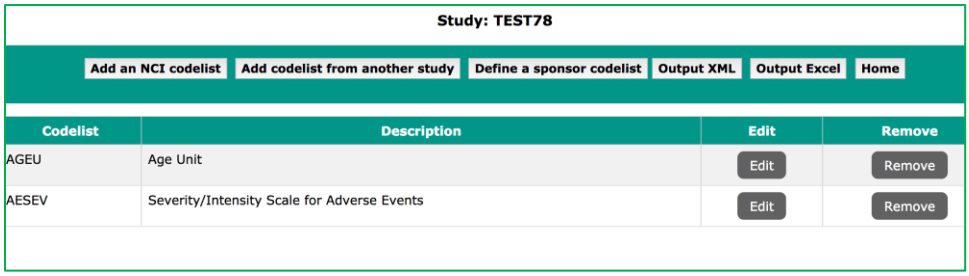
Upon choosing a codelist, a user is presented with a list of the terms contained in the chosen codelist, each with a checkbox next to it (see Figure 3). The user can subset simply by checking only the boxes next to the terms they want in the child they are defining. For extensible codelists, the user also has an opportunity to add extended terms. Once the user clicks the Save button, the new global codelist is added to the database, and the user is presented once again with the complete list of global codelists that now includes the child they just created.

Figure 3 – Creating a child codelist



A user can also define controlled terminology for a study. We start by creating a new study and adding to it codelists from the chosen NCI version (i.e. global codelists) (see Figure 4). For any one of the chosen codelists, the user can then create a child codelist in the manner described above at the global level. Such child codelists will only be associated with the study and not the global level. Users can also remove codelists from the study. Consider the UNIT example mentioned earlier. A user might initially add the full 640-term UNIT codelist to the study, then create all the child codelists from UNIT that are necessary, then remove the UNIT codelist. At any point during the process of defining a study's controlled terminology, a user has two options for output. One option is to generate Excel output. With the click of a button, a user can export all of the controlled terminology defined thus far for a study into Excel. In particular, the structure of the Excel file is exactly what is required by the Pinnacle 21 tool to upload metadata to create define.xml. Also, the click of another button produces XML output. In particular, the structure of the XML matches the structure found in define.xml (and ODM).

Figure 4 – controlled terminology for a study



WEB APPLICATIONS

A web application is an application in which a user enters the application through a URL entered into a web browser. From that point on the user engages in a sort of dialog with the application. The user enters information into an HTML form through text boxes, radio buttons, checkboxes, etc. Upon submitting the form, usually by way of clicking a button, the application responds by rendering a new web page, possibly a new form, based on the input provided by the user. For all of this to work, the following components are needed.

- A front-end user interface – This consists of HTML files that include CSS (Cascading Stylesheet) language for style and Javascript for interactivity within a page

- Server-side script – When a URL is requested, either by manually typing it or clicking a Submit button in an HTML form, this script executes code using as input the information provided by the user. The script uses a scripting language to process this input and react accordingly, sometimes by rendering another web page. The CTD application uses the Python scripting language.
- A web framework – This is the connection between HTML URLs and the script to be executed. It includes what is known as a *template language*, which allows us to build dynamic web pages (think of a SAS macro) by passing information from the script to the web form.

As a point of comparison, the first two of these components should not be entirely foreign to those who have developed applications in SAS. The interface is where the user initiates and interacts with the application. SAS does offer products such as AF® and SAS IntrNet® that offer interfaces. Within the Base SAS suite of tools is the WINDOW/DISPLAY functionality, available in the DATA step or in the macro language. Maybe the most familiar form of an interface is a simple SAS program that calls a macro. We can think of the script as the SAS code (macro or DATA step) that executes behind the scenes based on the user input.

A web application, like a SAS application, can process user input in different ways. In a simple application, the script (or macro) may simply handle input with scripting code. On the other hand, information about user input may be too much to handle only with code, and so may need to be managed in a database. A database is also useful if the application offers an opportunity to save information based on user input. For this reason, depending on the application, a database, as well as a query language for querying the database, can be considered the fourth and fifth components of a web application.

CTD is an application that allows users to define, output, and save codelists. While in theory we could keep information about each codelist and each of their terms in a program, best practice clearly says that this information is best managed in a database. The fact that we want to save the codelists we create only adds to the argument for a well-designed database. As mentioned earlier, the fact that the NCI codelists come to us in Excel may, on the surface, make Excel a tempting choice for a database, we've already covered the challenges of Excel as a database. SAS programmers might suggest a database of SAS data sets, but this could run into some of the same problems that Excel runs into. In addition, the proprietary nature of SAS would present challenges to the open source nature of the application. While traditional relational databases are available for free, we decided for this application to use a non-traditional *graph database* from Neo4j. This database, along with Cypher, the query language of Neo4j, becomes the fourth and fifth main components of CTD.

While reading the description of this application, a few questions may come to mind.

- What is the mechanism that allows my input on one web page to inform the contents of the next web page that appears?
- What versions of NCI codelists are available, and how are they made available?
- How does CTD know about the codelists defined by NCI?
- How does CTD know if a codelist is extensible or not?
- How does CTD know of the terms defined by a codelist?
- What does it mean for a codelist to be defined at the global level? Defined at a study level?

Throughout the remainder of this section, answers to these questions will become clearer as we discuss the five components mentioned above.

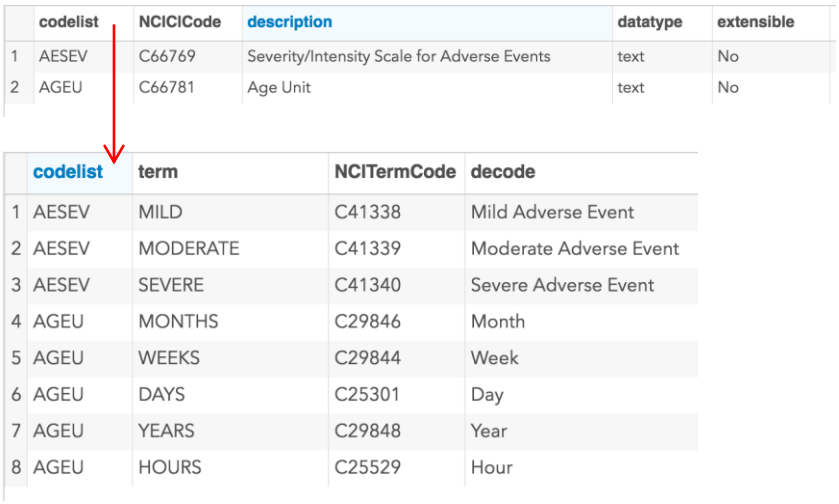
THE GRAPH DATABASE

When we think of a traditional relational database such as SAS or Oracle, we typically think of three dimensions – the column (or SAS variable), the row or record (or SAS observation), and the table (or SAS data set). We can think of a record as a collection of properties that describe a single observation, and the way these properties are related to each other on a given record is a natural part of the database. In a relational database, this is where the natural relationships come to an end. It's possible that different records, either within a table or across tables do have an inherent relationship to each other, but a relational database has no way to express that. A database designer may choose to express that relationship through the creation of variables (i.e. keys) whose values imply a relationship, but it's then up to a programmer to write code that captures the relationship (for example, through an SQL join).

The graph database also has the concept of a record as a collection of related properties. Such records are referred to as nodes. Nodes have labels, and the collection of all nodes with the same labels is the graph counterpart of the relational database table. What sets graph databases apart is the concept of relationships between nodes. A relationship between two nodes can be uni-directional, originating at one node and ending at another, or can be bi-directional. Like nodes, relationships also have labels, referred to as *types*, as well as properties.

We typically envision a relational database record as a row of related information, and a table as a collection of such rows, thereby painting a picture of a two-dimensional rectangular structure. Neo4j portrays nodes as points in space, and relationships between nodes as edges that connect them. Figures 5a and 5b illustrate the difference between the two, using the relationship between a codelist and the terms it contains.

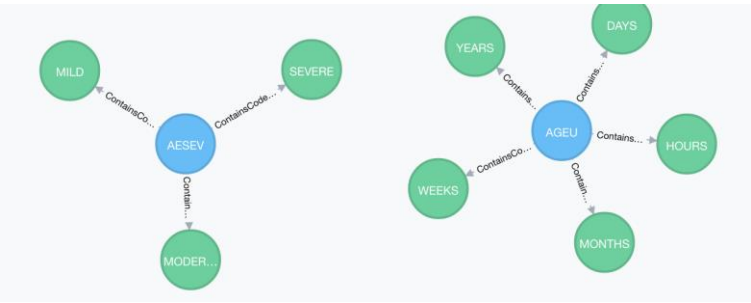
Figure 5a – relational database representation



	codelist	NCICICode	description	datatype	extensible
1	AESEV	C66769	Severity/Intensity Scale for Adverse Events	text	No
2	AGEU	C66781	Age Unit	text	No

	codelist	term	NCITermCode	decode
1	AESEV	MILD	C41338	Mild Adverse Event
2	AESEV	MODERATE	C41339	Moderate Adverse Event
3	AESEV	SEVERE	C41340	Severe Adverse Event
4	AGEU	MONTHS	C29846	Month
5	AGEU	WEEKS	C29844	Week
6	AGEU	DAYS	C25301	Day
7	AGEU	YEARS	C29848	Year
8	AGEU	HOURS	C25529	Hour

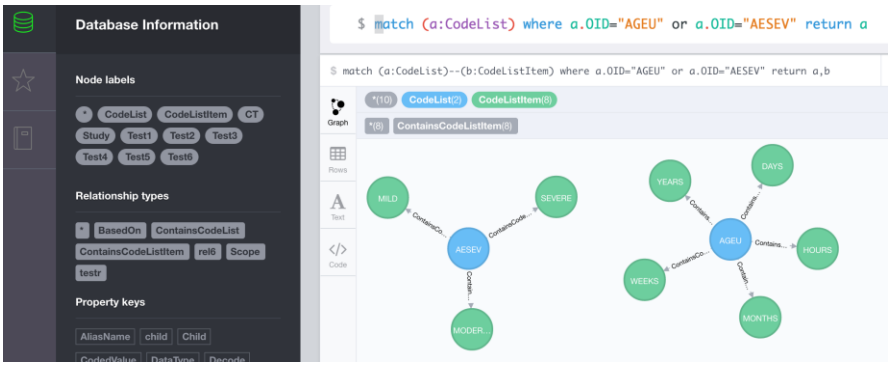
Figure 5b – Neo4j graph database representation



The distinguishing trait is that while we can see in figure 5a that values of one table match values in another, we can only infer that a relationship exists, and the only way to take advantage of it is for a programmer to write code to bring the information together. In the graph database, the relationship is an integral part of the database, and no inferences are necessary.

Neo4j makes available through its website a free version of its graph database. The download provides a development server that sits on your server or even on your laptop. Starting the server makes available a web interface at port 7474, from which you can interact with your database and execute queries. Upon download, a sample database as well as a brief tutorial is also available. See figure 6 below.

Figure 6 – Neo4j web interface

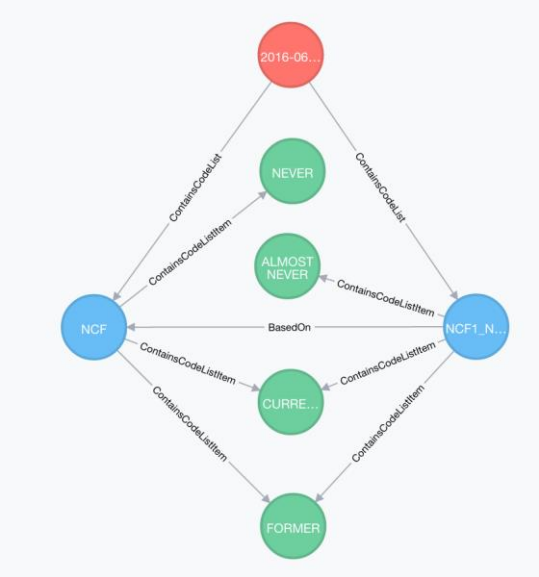


In CTD, the NCI-defined set of codelists is modeled as illustrated in figure 5b. Each codelist is represented with a node whose label is “CodeList”. Each of these nodes has properties that represent the codelist-level properties discussed earlier – data type, NCI code, whether or not the codelist is extensible, etc. Each codelist term is represented with a node labeled “CodeListTerm”, and each of these has properties representing the term-level properties, such as the term itself, a decode if applicable, the NCI code, etc. A uni-directional edge labeled “ContainsTerm” that emanates from the CodeList node and points to the CodeListTerm node connects them and expresses a relatoinship. If a term belongs to more than one codelist, then the node representing that term will have one edge for each such codelist directed at it. All codelists that belong to a particular version of an NCI publication are connected by an edge to a “Publication” node representing that version.

As the CTD user makes choices, behind the scenes CTD reacts by making changes to this database. For example, when a user creates a child codelist at the global level, a new CodeList node is added that is connected both to the Publication node that its parent is connected to as well as the parent codelist itself. It is also connected to a subset of the terms that its parent was connected to, as well as any new extensions defined. Figure 7 below illustrates an example. The red node is a Publication node, and attached to it are two CodeList nodes. One is the NCF codelist that is defined by NCI. The other is a

child codelist, created through CTD, from NCF. This is also connected to the NCF codelist through a relationship called “BasedOn”. The green nodes represent terms. The NCI codelist terms are “NEVER”, “CURRENT”, and “FORMER”. The child codelist is not connected to “NEVER”, but is connected to an extension term – “ALMOST NEVER”.

Figure 7 – The child codelist in the CTD database



When a CTD user creates a new study, a “Study” node is created that serves a role similar to that of the Publication node. As the user adds global-level NCI codelists to the study, edges are created between the Study node and those codelists. Child codelists created from these are created in a way similar to that described above, but are connected to the Study node rather than the Publication node. When a user chooses to remove a codelist from a study (think of the UNIT example above), the edge connecting it to the Study node is simply removed. When a user creates a custom codelist, a CodeList node is created and attached to the Study node.

CYPHER

Just as SAS provides us with a language for manipulating SAS databases, and more generally, SQL is a query language for relational databases, Cypher is the query language for Neo4j graph databases. In this paper we’ll cover only the very high-level basics of this language.

The main clause used for reading a database is the MATCH clause. This clause is used to specify patterns in the graph, and to return the desired elements of that pattern. In Cypher, the syntax for a pattern is meant to be a rough ASCII representation of how we view the database – nodes are represented with parentheses, and relationships are represented with hyphens and angle brackets to look like arrows. Relationship types are expressed inside square brackets, and filters can be expressed in curly braces or with a WHERE. The RETURN clause following a MATCH returns results from the query. While nodes and relationships can be returned, what is most useful to be returned for further processing is node properties. The following query returns the term, decode, and NCI code of all terms connected to a CodeList node whose OID value is “SEV”.

```
match (a:CodeList {OID:”SEV”}) – [b:ContainsTerm] -> (c:CodeListTerm)
return c.term as CodedValue,c.decode,c.NCICode
```

In the MATCH clause above, we are trying to match all patterns that start at any CodeList nodes that have an OID property whose value is SEV, proceed through a relationship called ContansTerm, and end at a node called CodeListTerm. Note that in the specifications of the nodes and the relationships, the label is preceded by a colon – this is required by the syntax. Preceding that is an optional identifier – in this case, a, b, and c. These allow us to make references to these nodes and relationships later, as we did in our RETURN clause. The RETURN clause works much like SELECT in SQL, where we name the properties we want returned in our results, optionally giving them an alias name as we did with c.term. The query results look much like a SAS data set or any other relational database table. Each returned property becomes a "column", and one "row" is returned for each CodeListTerm node that was matched.

With the example above and the discussion thus far, we have merely scratched the surface of the pattern matching and result return capabilities of Cypher. Patterns can involve many more nodes and relationships, arrows can point both to the left and to the right, even for the same node, we can filter on the length of a pattern, we can use the UNION operator, we can sort our results, and many more.

The CREATE clause creates patterns, or individual nodes or relationships within existing patterns. In the following example, a pattern consisting of three nodes (plus properties) and two relationships is created.

```
create (a:Study {name: "MyStudy"}) - [:ContainsCodeList] -> (b:CodeList: {OID: "CustomList"}) - [:ContainsTerm] -> (c:CodeListTerm {Term: "CustomTerm"})
```

In the following, we create a relationship between two already-existing nodes.

```
match (a:CodeList),(b:CodeListTerm) where a.OID="ChildCodeList" and b.Term="Mild" create (a)-[:ContainsTerm] -> (b)
```

In this example, we use WHERE as an alternate way to filter. Note that when the nodes already exist, we first have to match them, then create the relationship between them.

Up to this point the only environment we've discussed for writing and executing queries is the interactive web interface. Keep in mind though that in CTD, as is the case with most web applications, the user's only interface is a set of HTML forms, and that the manipulation of the database takes place behind the scenes as a response to the user input. This typically happens by way of a server-side program that acts as a liaison between the interface and the database. With the remainder of this paper, we'll first see how the Python scripting language can issue queries to the Neo4j database, and we'll conclude by seeing how a Python program is connected to the interface.

CONNECTING PYTHON TO NEO4J

In this paper we won't get into any kind of Python tutorial, but we will talk enough about it to get a feel for its role in CTD. Python has object-oriented aspects to it. If you're not familiar with object-oriented programming, don't worry about the details – you should be able to follow along.

For those who have spent a career working with the SAS programming language, SAS can be thought of as a closed system. The Base SAS programming language, or more specifically, the Base SAS processor, is one of many products sold to SAS customers for an upfront price, plus an annual licensing fee. We can write SAS code with any text editor, but the licensing fee allows us to execute it. It allows us access to data in SAS's proprietary data set structures. SAS applications written with the macro facility can only be used by those who have the license.

Python, as an open-source product, is just the opposite. The Python processor, also known as the *interpreter*, is available as a free download from the Python website. Python is composed of modules – code units containing functions and classes that each serves a specific purpose. The Python interpreter ships with a standard library of modules, and modules written by third parties are available for free download and installation. Python doesn't have proprietary data structures like the SAS data set, but modules are available to work with many database structures. The Python module *py2neo*, and more specifically, the Graph class inside py2neo, make connection to a Neo4j database possible. After downloading and installing this module, the IMPORT and the FROM operators make data accessible to the program.

```
from py2neo import Graph
graph = Graph()
```

The second statement creates an instance (or object) of the Graph class called *graph*. In short, this is what establishes the connection with the database. The fact that no parameters are specified with the Graph class means that the connection, by default, is to port 7474 of localhost. With this object, we can execute the EXECUTE method whose argument is a single Cypher statement that is immediately executed against the database.

Here's where things get a little unusual to a SAS programmer. Methods in any object-oriented language return some kind of value, and to capture that value we typically store it in a variable. But as we showed above, the result of a query is, like a SAS data set, a two-dimensional data structure. More specifically, the EXECUTE method returns an object referred to as a *RecordList*. This means that in the example below, the value of the variable *results* is a structure like that of a SAS data set!

```
results = graph.cypher.execute('match (a:CodeList {OID: "SEV"})-
[:ContainsTerm]->(b:CodeListTerm) return b.Term,b.Decode,b.NCICode')
```

We know that in SAS we can read a data set with a SET statement. Python doesn't have a SET statement to read RecordList objects, but we can define an iterative loop to cycle through the "rows", or more accurately, the *Records* that make up the RecordList (similar to what the SET statement is doing behind the scenes in the SAS DATA step). Just as the DATA step has the DO loop, Python has the FOR loop that allows us to process a RecordList one Record at a time.

```
for x in results:
      print "Term: " x[0]
      print "Decode: " x["1"]
      print "NCI Code: " x["2"]
```

With each iteration of the loop, x represents a new record, and x[i] represents the (i+1)st column returned. The above code simply prints the values of each of the properties from each of the CodeListTerm nodes.

We now have a means for executing queries against a Neo4j database from a Python script. For the last piece of the puzzle, we now put it all together with a web framework.

CONNECTING PYTHON TO THE USER WITH THE DJANGO WEB FRAMEWORK

Django is a web framework that allows us to connect web forms to scripts. Although Django, and web frameworks in general, offer much more as a way to get your web application off the ground, in this section, we'll provide a brief explanation specifically of how CTD uses Django to pass information from a

Python script (e.g. information collected from a database) to a web form, and information collected from a user in a web form back to the script. We will illustrate with a simple example.

When working with NCI controlled terminology, building either global child codelists or study codelists, one of the first questions that needs to be asked of the user is which publication version the user is interested in. Of course the user can only choose from a list of versions that are in the current database, so the web page provides a dropdown menu of those choices. One way to do this would be to hard-code those choices into the HTML, but that would require changes to the program as new versions become available in the database. Instead, the developer chooses to query the database for all such versions, and pass them into the web page's dropdown menu. We saw in the last section how to query the database. The Python code might look something like this:

```
versions = graph.cypher.execute("match (a:Publication) return a.version as version")
```

This query returns what looks to SAS programmers like a one-variable data set, where the variable contains one row for each NCI version in the database, but is stored in the Python variable "versions". The next question is, how do we pass this into a dynamic HTML file.

In this paper we'll skip all of the details regarding the setup of Django – downloading the development server, configuration files, etc, - but we will briefly mention here the URL configuration file. This is a short Python program that maps requested URLs to Python functions that return web pages. When a particular URL is requested, Django finds that URL in the configuration file, determines the associated function, and executes that function. Part of that function must include Python code that renders a web page.

```
return render(request, 'nextwebpage.html', {'AvailableVersions':versions})
```

Think of this as a SAS macro call with parameters. We'll ignore the first parameter to the render method (i.e. *request*) above and concentrate on the second and third. The second parameter simply indicates which web page is to be loaded. The third parameter is a bit unusual. It's an optional parameter, enclosed in braces, that allows us to pass to the HTML form a *dictionary.* A dictionary is simply a set of name-value pairs, where the value of any of the pairs is referred to by its corresponding name. In this case, we have passed one pair with a name set to *AvailableVersions*, and the value we're passing to it is *versions*, the result of our database query. We can now make reference in our HTML file to AvailableVersions.

SAS programmers, especially those that like to build macros, should fully appreciate the way Django can pass information from a Python program to a web page. If the Render method above is like a macro call, then the HTML file is like a macro program. As we know, the macro's job is to generate SAS code. The macro is therefore a combination of literal hard-coded text to be generated, text generated as a function of macro logic, and resolved macro variables. For example, consider the following macro:

```
%macro demo(name) ;
Hello
%if &name=Jerry %then Dad ;
%else &name ;
%mend ;
```

In this macro, "Hello" is literal text that is generated unconditionally, without any macro logic when the macro is called. On the other hand, what is generated afterward is done so with logic, and as a function of the macro parameter. The logic is built with language-specific keywords and syntax, such as "%if" and

“%then”, as well as variable syntax such as “&name”. What is known as the *Django template language* does the same thing for HTML files.

One way to create a dropdown list in HTML is through a SELECT tag, with OPTION tags nested inside.

```
<select name = "CTVersions">
      <option>Choice 1</option>
      <option>Choice 2</option>
      <option>Choice 3</option>
</select>
```

In SAS, you can imagine how to cycle through a list to iteratively generate text with %DO. The Django template language is similar.

```
<select name = "CTVersions">
{% for x in AvailableVersions %}
      <option>{{ x.version }}</option>
{% endfor %}
</select>
```

Just as with %DO, the *for* in the {% %} grouping creates an iterator, x, that iterates through each record in the record list (i.e. query result) that was passed to the template under the name AvailableVersions. Keep in mind that this record list contained a variable called *version*. x.version is the value of version in the xth iteration. This, as well as the double braces surrounding its reference, remind us of SAS’s macro variable. The loop terminates with *endfor,* again inside of {% %}.

Of course, when the form is submitted, the choice made by the user must be captured by the script. When an HTML form is submitted, the URL named in the form tag (<form>) is requested, according to the method (GET or POST) specified in the same tag. When this happens, as described above, the URL is matched with a Python function according to the URL configuration file, and the corresponding function is executed. When this happens, every element in the form that was submitted that has a *name* attribute is passed to the script in the form of a dictionary, where the value of any given pair is the value for the corresponding *name* attribute in the web page. So if the form that contained the HTML above is submitted to a URL called “nextURL” using the GET method having chosen 2016Q4 among the NCI versions, then this information 2016Q4 is the value of a name-value pair whose name is CTVersions. The example below shows how this can be captured in the script and assigned to a Python variable called ChosenVersion.

```
ChosenVersion = request.GET["CTVersions"]
```

We now know which version the user wants, and we can refer to ChosenVersion when we query the database for codelists under this version.

```
ChosenCodeLists = graph.cypher.execute("match (a:Publication {version: '" +
ChosenVersion + "'} - [:ContainsCodeList] -> (b:CodeList) return b.OID")
```

CONCLUSION

CTD brings the clinical intelligence to the process of defining controlled terminology that Excel could never bring. The combination of a scripting language that reacts to user input through a web browser, and a database with controlled access, and a web framework that brings it all together saves the user

time otherwise wasted on administrative Excel tasks such as copying/pasting, adding new rows and columns, etc, thereby allowing the user to concentrate solely on the clinical task at hand. At the same time it enforces the clinical rules built into it, ensuring compliance and accuracy. The scope of CTD – controlled terminology – may seem limited, but we can imagine applying similar technology to other segments of the clinical trial metadata flow, such as the intelligent creation of programming specifications and define.xml. Hopefully you have seen from this paper that by combining our SAS knowledge with some outside-the-box thinking, we can open up a new world for developing and processing clinical metadata.

CONTACT INFORMATION

Mike Molter
Wright Avenue Partners
919-414-7736
molter.mike@gmail.com